**MASTER EN SCIENCES INFORMATIQUES**

# Etude comparative de l'évolution de systèmes d'information

Hainaut, Jérôme

*Award date:*
2019

*Awarding institution:*
Universite de Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

# Etude comparative de l'évolution de systèmes d'information

Hainaut Jérôme

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

# Remerciements

L'écriture de ce mémoire fut une expérience enrichissante mais difficile. Beaucoup de personnes m'ont aidé à faire de ce mémoire ce qu'il est aujourd'hui. Travailler tout en écrivant un mémoire n'est pas une tâche simple. Sans aide, je ne sais pas si j'aurais réussi à le finir.

J'aimerais commencer par remercier mon promoteur, le Professeur Anthony Cleve, sans qui ce mémoire n'existerait pas. Non seulement, il a accepté ma proposition de mémoire, mais en plus il m'a soutenu jusqu'au bout. Il fut disponible pour m'aider à avancer et m'a débloqué à de nombreuses reprises. Ses conseils et ses retours furent précieux pour mon avancement.

Une autre personne en particulier me donna la force pour aller jusqu'au bout de cette formidable aventure. J'aimerais remercier Maëlle qui me donna le courage nécessaire pour finir mon master en cours du soir. Je désirerais aussi la remercier pour son soutien et son aide sur l'écriture de mon mémoire.

J'aimerais également remercier toutes ces personnes avec qui j'ai eu l'occasion de discuter de mon mémoire. Ces discussions m'ont donné des ressources précieuses pour enrichir mon mémoire avec un point de vue de praticien.

Enfin, j'aimerais remercier ma famille pour son soutien dans mes études en général. Un merci particulier à ma maman, Geneviève, pour son aide dans la correction de mon orthographe.

# Abstract

Comment faire évoluer notre application ? Quand nous travaillons sur un logiciel informatique, nous sommes confrontés à cette question au quotidien. Cette question est simple, mais la réponse ne l'est pas.

Il existe de nombreux ouvrages et outils pour aider à construire notre application. Ceux-ci sont, pour la plupart, spécialisés dans un domaine en particulier (rétro-ingénierie, analyse d'impacts, migration de la base de données, aide à l'évolution du code ...). Il est difficile de savoir ce qui existe ou ce qu'il reste à faire.

Le but de cette recherche est d'aider à répondre à la question de l'évolution en donnant des pistes de réflexion. Pour ce faire, nous avons regroupé et comparé des ouvrages et outils permettant d'avoir de l'aide lors de l'évolution d'une application.

# Sommaire

# Table des figures

# Introduction

"*Remember, code is your house, and you have to live in it.*" [1]

Le code et la base de données évoluent tout le temps. Que ce soit pour ajouter une fonctionnalité, corriger un bug, améliorer l'architecture (le design) ou encore pour optimiser l'utilisation des ressources [36]. Peu importe la raison, le processus reste le même, il passe par la modification du système pour répondre à de nouvelles demandes.

Ces modifications sont tantôt simples, tantôt complexes. Dans les deux cas, il est possible d'apporter les changements de multiples manières, certains rendant les futures modifications plus simples et d'autres les rendant plus compliquées.


FIGURE 1 – Les impacts d'une nouvelle fonctionnalité [36]

De mauvais choix consécutifs engendrent une application qui évolue mal. Au cours des développements, le code et la base de données deviennent de plus en plus complexes et il devient de plus en plus difficile de les maintenir. Ils deviennent de plus en plus désordonnés surtout si aucune attention n'a été mise en place pour limiter le désordre. Celui-ci entraîne aussi une diminution de la productivité des programmeurs.

Dans la littérature, nous retrouvons le terme "*legacy*" pour une application où tout changement devient un véritable enfer.

Faire évoluer l'application correctement n'est pas chose aisée. **M. Feathers** [36] nous dit que "*Even the most disciplined development team, knowing the best principles, using the best patterns, and following the best practices will create messes from time to time*". Nous pouvons dire que, faire évoluer une application, c'est compliqué.

Les causes pour qu'une application tourne mal sont nombreuses :

1. Michael Feathers. *Working Effectively with Legacy Code.* Prentice Hall Professional, 2004, page 58.

— une jeune équipe : celle-ci, n'ayant pas le recul nécessaire, fait des erreurs ;
— l'utilisation d'une technologie inadéquate à la problématique : certaines sociétés se sont vues revenir en arrière après s'être "jetées" sur les bases de données NoSQL [32] ;
— la pression de la direction : devoir aller vite, toujours plus vite, pour délivrer de la valeur business sans prendre le temps de faire les choses correctement [36, 51] ;
— l'historique du système : travailler dans un système existant force parfois à devoir faire la même chose même ci cela n'est pas "propre" [36] ;
— pas de vision d'ensemble : si chaque partie du code est bien pensée, cela ne veut pas dire que l'architecture soit correcte.

Nous avons pu, dans notre pratique, constater ce que veut dire une application qui tourne mal. Dans l'un des projets sur lequel nous avons travaillé, le résultat des choix pris tout au long du développement et de la maintenance de l'application a entraîné une situation que nous qualifierons de catastrophique.

Le projet consiste en une application monolithique composée principalement d'une base de données et du logiciel y accédant. Le projet a commencé, au début de l'orienté objet, avec une équipe jeune qui ne maîtrisait pas les concepts utilisés. Dans un même temps, la direction a forcé l'utilisation de technologies inadaptées. Beaucoup d'équipes se sont relayées sur le projet apportant chacune leur propre manière de travailler. Le résultat de cette histoire quelque peu mouvementée fut une application où le temps pour apporter un changement se compte en semaines, en mois, voire en années pour les changement les plus conséquents.

En travaillant sur ce projet, nous avons commencé à nous poser des questions quant à la manière de faire évoluer des applications. C'est la raison principale qui nous a amenés à effectuer la présente recherche.

Au plus une application est complexe, et les applications ont tendance à le devenir de plus en plus [49], au plus il devient difficile de la faire évoluer sans risque[51]. Maintenir une application veut dire garder synchronisés le code, la base de données et son schéma tels que l'on peut le voir sur la Figure 2. Une étude montre que quelques modifications dans le schéma résultent en 100 à 1000 lignes impactées dans le code [50].

Sur base de ces constatations, nous nous sommes posé la question suivante : comment garder la consistance d'une application au cours de son évolution, aussi grande soit-elle ?

Pour savoir comment faire évoluer une application, il faut déjà savoir ce qu'on entend par la faire évoluer.

Pour illustrer l'évolution d'un logiciel, prenons le schéma simplifié en Figure 2. Nous avons, à gauche, la représentation initiale de l'application. Nous avons un "Schéma" qui représente le modèle de la "Base de données". La "Base de données" interagit avec le "Programme" offrant aux utilisateurs les fonctionnalités. Cette application est celle que nous voulons faire évoluer. Nous constatons que, dans le cas d'utilisation de bases de données NoSQL, il n'y a pas la distinction entre schéma et base de données car les bases de données sont "*Schemaless*".

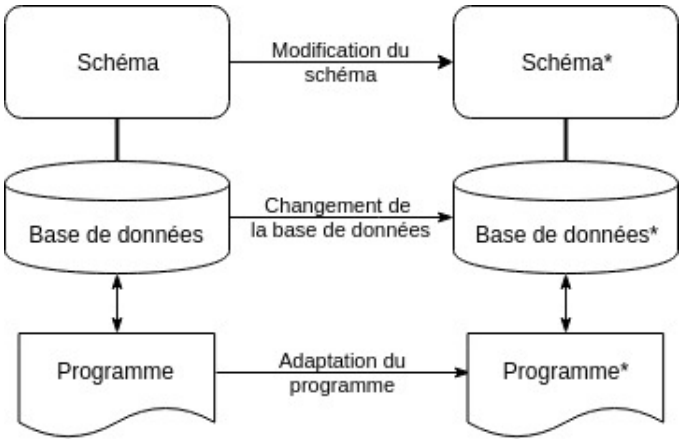


FIGURE 2 – évolution des logiciels [50]

De l'autre côté, nous avons l'application après modification (avec les "*" sur le schéma). Entre les deux, nous avons les opérations réalisées pour arriver à ce résultat.

Nous remarquons qu'avec plusieurs bases de données et/ou programmes, la représentation ci-dessus reste similaire. Si nous avons plusieurs bases de données, nous pouvons découper le problème base de données par base de données. Il en va de même si nous avons plusieurs programmes.

Compte tenu de ce qui précède, nous avons une idée de ce que veut dire "faire évoluer une application". Nous savons que nous voulons passer de l'application actuelle vers la nouvelle application. Pourtant, nous ne savons toujours pas comment le faire.

Il existe de nombreux travaux expliquant comment faire évoluer des logiciels orientés données avec bien souvent des outils permettant de le faire de manière plus ou moins automatique. Chacun de ces travaux et outils traite bien souvent d'un point précis tel que la migration d'une base de données relationnelle vers une base de données NoSQL [46, 62, 64, 66]. D'autres couvrent de manière plus large la question en donnant, par exemple, un résumé de la rétro-ingénierie des données [35].

Peu de recherches couvrent de manière transversale la question de l'évolution d'une application orientée données. Afin de mener à bien notre recherche, nous allons procéder à un état de l'art. Pour ce faire, nous allons, autant que possible, recenser la littérature scientifique, les outils en relation avec le sujet de la présente recherche et toutes autres sources pertinentes, puis les analyser et les comparer.

Avant d'aller dans le code et le changer, il est nécessaire de savoir ce que fait l'application courante. Savoir comment fonctionne le logiciel nous apporte des réponses pour pouvoir identifier les parties à changer. C'est la rétro-ingénierie

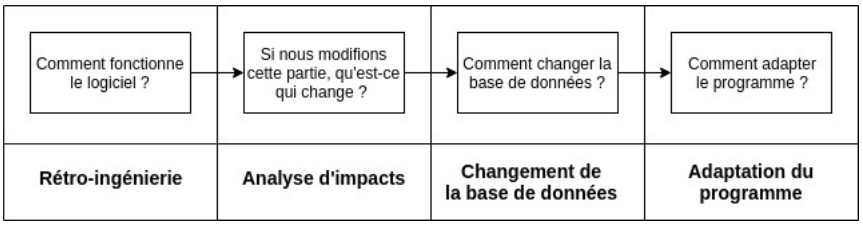


FIGURE 3 – Les étapes de l'évolution d'un système

logiciel. La **rétro-ingénierie** (chapitre 1) permet de comprendre comment fonctionne le système. Avant d'apporter une modification, il faut connaître les **impacts** (chapitre 2) qu'elle peut engendrer. Il est nécessaire d'identifier les parties impactées pour pouvoir répercuter la modification partout où cela est nécessaire. Le but est de faire en sorte que l'ensemble de l'application fonctionne toujours correctement et de manière cohérente. Une fois tout ce travail préalable terminé, nous pouvons changer la **base de données** (chapitre 3) et le **programme** (chapitre 4) pour arriver au résultat attendu.

# Chapitre 1

# Rétro-Ingénierie

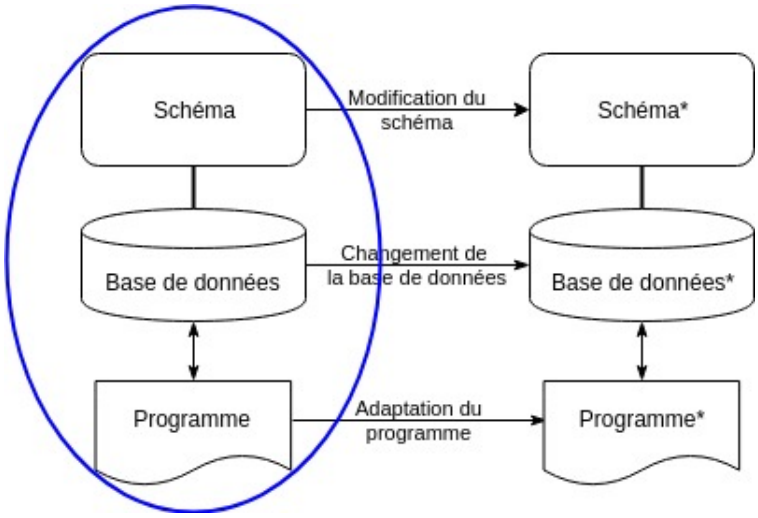


FIGURE 1.1 – évolution des logiciels, rétro-ingénierie

Une partie des développeurs pense que prendre le temps de comprendre le code, ce n'est pas travailler [36]. Dans ce chapitre, nous traiterons de la rétro-ingénierie. Nous mettrons entre autres en avant ce que c'est et pourquoi il est utile de comprendre le code mais aussi l'architecture du logiciel ainsi que ses données.

Nous allons nous focaliser sur le système existant (voir Figure 1.1).

Dans la section 1.1, nous donnerons une définition de la rétro-ingénierie et le principe de base de son fonctionnement. Puis dans la section 1.2, nous expliquerons pourquoi la rétro-ingénierie est réalisée. Ensuite, nous détaillerons dans la section 1.3 un ensemble de techniques et méthodes pour faire de la rétro-ingénierie. Enfin, dans la section 1.4, nous prendrons du recul sur les approches proposées à l'aide d'un tableau récapitulatif des approches de la rétro-ingénierie.

## 1.1 Définition et fonctionnement

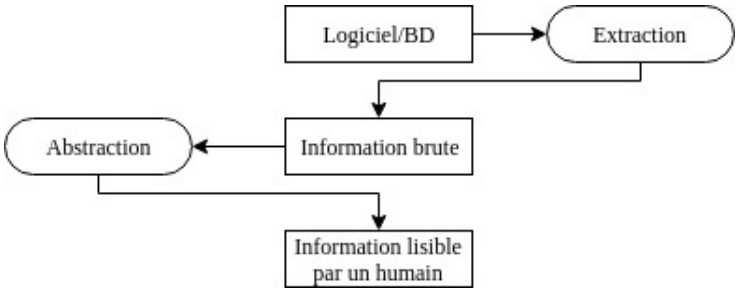


FIGURE 1.2 – Le principe de la rétro-ingénierie

La rétro-ingénierie, en anglais *reverse engineering*, est un processus qui permet de comprendre la structure et les interactions d'un système. C'est documenter un sujet afin d'en faciliter sa compréhension [35]. Nous pouvons le définir comme l'examination d'un processus déjà implémenté afin d'en comprendre son fonctionnement. Pour le représenter sous une autre forme, nous pouvons dire que la rétro-ingénierie logiciel permet de corriger, mettre à jour, récupérer ou réécrire une partie ou l'entièreté d'un système [26, 27]. Elle répond aux questions "Qu'est-ce ?", "Comment ça marche ?", "Qu'est-ce que ça ne fait pas ?".

A l'origine, la rétro-ingénierie, dans le domaine du logiciel, s'appliquait uniquement aux données. Il est question dès lors de *data reverse engineering*. Nous pouvons lire **Chikofsky** [29] la définissant comme étant *"a collection of methods and tools to help an organization determine the structure, function, and meaning of its data"*. (en français : "une collection de méthodes et d'outils pour aider une organisation à déterminer la structure, le fonctionnement et la signification de ses données.")

Ne confondons pas données et base de données. La rétro-ingénierie des données concerne tout ce qui touche aux données [35]. Si la base de données est un élément majeur, ce n'est pas le seul. Les mails ou encore les fichiers sont des éléments importants contenant une grande part de la connaissance de l'application.

La *data reverse engineering* a donné lieu à de nombreuses recherches et a été généralisée en *reverse engineering* (rétro-ingénierie) au cours du temps [35].

Comme représenté sur la Figure 1.2, la compréhension du système, quel que soit le chemin utilisé, va fondamentalement toujours fonctionner de la même manière [27].

1. **Extraction** des données brutes. Nous collectons toutes les informations nécessaires pour comprendre le système.
2. **L'abstraction** de celles-ci. Nous mettons en forme les données récoltées pour qu'elles soient lisibles par un humain.

Dans le cas particulier de la rétro-ingénierie sur une base de données, le fonctionnement se traduit par l'extraction du schéma physique brut vers son abstraction en un schéma conceptuel.

Il existe quatre techniques d'analyses pour la rétro-ingénierie, l'analyse manuelle, statique, dynamique et historique [27]. L'analyse statique fut la première à être utilisée après la manuelle car plus simple à mettre en œuvre. L'analyse dynamique, quant à elle, est plus complexe, plus coûteuse et incomplète. Elle permet toutefois d'apporter un bon complément à l'analyse statique car elle apporte un autre point de vue. L'analyse historique, quant a elle, n'est possible que depuis l'apparition des outils de gestion de versions.

## 1.2 Destination

Le but principal de la rétro-ingénierie est d'apporter une vue plus claire du système pour un humain [27]. Elle est utile de multiples façons et la recherche n'a cessé de réaliser des progrès depuis ses débuts dans les années 70 [27, 35, 19]. C'est entre autres grâce à elle qu'il a été possible d'éviter des milliards de pertes lors du passage à l'an 2000 [27].

La rétro-ingénierie est à la base de beaucoup d'autres approches. Par exemple, si vous voulez connaître l'impact d'un changement (chapitre 2), comment faire si vous n'êtes même pas capables de dire ce que fait l'application ? De même, si vous voulez identifier des patterns [63] pour de futures utilisations, la rétro-ingénierie est essentielle. Elle vous permet de comprendre le système et donc le milieu dans lequel vous récupérez les patterns.

Il existe bien d'autres raisons d'utiliser la rétro-ingénierie :
— maintenir le système (Comprendre avant d'ajouter) ;
— le réécrire (refactoring) ;
— l'étendre (ajout de fonctionnalités) ;
— migrer (partiel ou total) ;
— l'extraction de résultats (statistique) ;
— réutilisation ;
— ...

Il n'est pas facile de garder la connaissance de l'application à jour. Les compagnies ont toutes la problématique de maintenir, remplacer ou gérer partiellement ou complètement leurs logiciels [26]. Pour apporter une modification au système, il est vital de comprendre ce qu'il fait, d'identifier son comportement, pour que le changement soit approprié.

Ces changements peuvent se situer à différents niveaux. Ils peuvent impacter l'architecture. Ils sont peut-être sur une correction des règles d'une fonctionnalité. Ils impliquent potentiellement tout un module. Il est aussi possible qu'ils soient liés aux traitements des données. Dans tous ces cas, la rétro-ingénierie est une étape importante qui facilite la compréhension du système afin de le garder cohérent.

Il n'est pas rare qu'avec l'évolution constante de l'application apparaissent des symptômes dûs à l'âge [25]. Michael Feathers [36] appuie ce point en expliquant que les applications qui durent longtemps ont tendance à mal évoluer même si, à la base, celles-ci étaient bien pensées. Les facteurs sont multiples, que ce soit par passage successif de développeurs, des pressions extérieures, l'évolution des technologies, ...

Un autre problème causé par l'âge grandissant d'une application est le maintien de l'architecture.

La compréhension de l'architecture du système a aussi une grande influence pour garantir la longévité du logiciel. Elle facilite la maintenance de l'architecture [27]. Pour garder cette connaissance de la vue d'ensemble, les organisations utilisent le rôle d'architecte [36]. Si celui-ci joue un rôle crucial, seul, il ne sait garantir ni la maintenance de l'architecture ni son évolution correcte. C'est en travaillant avec l'équipe qu'il est possible de garder une cohérence dans l'application. Si ce travail n'est pas réalisé, cette architecture peut se dégrader petit à petit.

Quand les équipes ne sont pas conscientes de l'architecture de leur système, il tend à se dégrader [36]. Cette dérive peut provenir des facteurs suivants :

— le système est tellement complexe qu'il n'y a pas de vue d'ensemble ;
— le système est tellement complexe que cela prend longtemps pour avoir une vue d'ensemble ;
— l'équipe est dans un mode d'urgence si bien que la vue d'ensemble est perdue.

Nous avons pu vivre dans un projet de grande ampleur les conséquences de l'absence d'une vue d'ensemble du système. Selon que nous soyons dans une partie de l'application ou une autre, rien n'était pareil. Il n'était pas rare de parcourir des classes de plus de 6000 lignes de code contenant toutes les couches logiques. De multiples frameworks pour une même problématique étaient utilisés. Il nous est même arrivé de trouver du code où deux technologies différentes étaient utilisées au même endroit pour répondre à un même problème.

Avec l'absence de maintien de la connaissance de l'application, nous arrivons à des systèmes dit *legacy*. Il n'est pas rare que ceux-ci contiennent des parts entières de la connaissance de l'organisation. De plus, ils sont souvent si gros et offrant des fonctionnalités si critiques qu'il est très coûteux et très risqué d'en développer un nouveau.

Ces systèmes arrivent à un tel point que les personnes ne connaissent que quelques places spécifiques où le système a été hacké afin d'amener les modifications demandées. Par exemple, ce framework maison qui est là depuis l'origine de l'application et qui n'a pas été maintenu depuis plusieurs années.

Pour les entreprises, il est crucial de récupérer et comprendre tout ce que renferment ces systèmes pour leur bon développement. [25]

Pour éviter de tout jeter et de recommencer à zéro, il est nécessaire de faire une analyse complète de ces applications *legacy*. La rétro-ingénierie se focalise principalement sur ces systèmes *legacy* qui font tourner des opérations critiques pour les compagnies [26].

Comme nous le disions au début de cette section, la rétro-ingénierie est là pour offrir une vue du système compréhensible par les humains. Son objectif est de donner une autre vue du système. Parmi ces vues, nous avons les diagrammes [15, 17, 18], les métriques [16], les graphiques, les rapports, ...

## 1.3 Approche existante

Au cours de l'histoire de l'informatique, il y eu beaucoup de recherches différentes sur la rétro-ingénierie. **Davis et Aiken** [35] ont parcouru tout un pan de cette histoire jusqu'en 2000. Ils se sont focalisés sur la rétro-ingénierie des données. Là où **Canfora et Penta** [27] nous en apprennent plus sur ce qui s'est passé pour la rétro-ingénierie sur le code et l'architecture.

Un long chemin a été parcouru depuis les débuts où tout était fait à la main. **Davis et Aiken** [35] nous expliquent qu'au départ, comme dit plus haut, la recherche s'est principalement orientée vers les données. Les recherches de l'époque donnèrent les premiers algorithmes de traduction avec la création de diagrammes entité/relation. Ces algorithmes utilisaient, comme source, un modèle de données relationnelles spécifique voire des fichier plats. Afin de généraliser les algorithmes de traduction, des méthodologies ont vu le jour.

Avec l'ensemble de ces méthodes, arrivent les premiers outils automatisant la création des modèles entité/relation sur base de fichiers plats ou de bases de données relationnelles. L'arrivée et l'utilisation des outils de gestion de version ont permis l'utilisation de l'historique des modifications pour améliorer les résultats.

De même, la rétro-ingénierie s'est principalement focalisée sur les logiciels procéduraux. C'est tout naturellement qu'elle s'est dirigée vers l'orienté objets en commençant par l'extraction d'objets dans les logiciels procéduraux [27].

Avec l'élargissement des applications et de leurs bases de données, il a été nécessaire de trouver des stratégies pour que la rétro-ingénierie puisse se faire dans des délais raisonnables. Des approches de découpe ("*slicing*" en anglais) ont, pour ce faire, été apportées, permettant de répondre plus facilement et plus efficacement au problème de la rétro-ingénierie actuelle.

Dans la suite de cette section, nous verrons sur trois axes et de manière non-exhaustive ce qui existe aujourd'hui pour répondre à la problématique de la rétro-ingénierie. Ces trois axes sont : l'architecture (sous-section 1.3.1), la base de données (sous-section 1.3.2) et le code (sous-section 1.3.3).

### 1.3.1 L'architecture

L'architecture est la vue d'ensemble du système. La connaître ne garantit pas la bonne évolution du système mais en facilite grandement sa maintenance. Supposons que nous ayons une application qui est découpée en modules par groupes de fonctionnalités. Si vous devez en ajouter une, où allez-vous l'ajouter ? Si vous avez la vue d'ensemble de l'application, même si rien ne certifie que tout sera parfait, vous pourrez au moins ajouter cette nouvelle fonctionnalité au bon endroit.

Par expérience, sur une application *legacy*, le simple fait de savoir où apporter une modification est loin d'être garanti. Il n'a pas été rare qu'il nous ait fallu deux à trois jours de recherche pour apporter une correction, simplement parce que personne ne connaissait le point d'entrée de la fonctionnalité.

C'est là que le *"Model Driven Reverse Engineering"* (MDRE) entre en jeu. C'est une manière de récupérer la connaissance sur l'application [26]. Le principe consiste en la génération de diagrammes sur base du code source. Avec ce genre d'approche, il est par exemple possible de générer un diagramme de classe de l'application sur laquelle nous travaillons.

Il existe plusieurs outils de ce genre. EasyCODE, Staruml ou encore Visual-paradigm [15, 17, 18] sont des outils qui permettent entre autres de générer des modèles UML sur base du code. Il est possible aussi d'avoir la génération de diagrammes directement intégrée dans les IDEs.
Jetbrains [8] propose cette fonctionnalité dans certaines versions de leurs outils. Directement intégrée, il est possible de naviguer immédiatement entre le code et les diagrammes.

**Brunelière et al.** [26], quant à eux, proposent un plugin eclipse pour faire du MDRE. Ce plugin eclipse (MoDisco) est un framework offrant un méta modèle générique qui rend possible l'ajout de plugins à celui-ci. Ils sont partis du constat que la plupart des solutions sont spécifiques au langage. Ils ont voulu utiliser une méthode plus générique applicable à plusieurs langages.

**Michael Feathers** [36] nous propose une toute autre approche. Se basant sur le fait que certains développeurs connaissent déjà une partie de l'application et donc son architecture, il est alors possible de retrouver la connaissance de celle-ci.
En utilisant sa solution qu'il nomme "*Telling the story of the system*", il est réalisable d'extraire une vue simplifiée de l'architecture. Cette technique consiste en l'explication de l'architecture de l'application avec des mots simples. De cette manière, il est possible de mettre en évidence les principaux problèmes. Quand dans l'explication nous sommes obligés de "mentir" pour rester simple, c'est un signe d'un problème potentiel dans l'architecture du programme. Avec cette méthode, il est même faisable d'expliquer l'architecture que nous voulons mettre en place avant de développer. Cela nous permet d'anticiper une bonne ou une mauvaise construction architecturale du système.
Dans son livre, il propose d'autres solutions basées sur l'expérience acquise des développeurs sur l'application et son architecture. Il se réfère également à l'ouvrage "*Object-Oriented Reengineering Patterns*" qui apporte d'autres solutions.

### 1.3.2 La base de données

Un point important de l'application est sa base de données. La comprendre est essentiel pour pouvoir la faire évoluer tout en gardant les données cohérentes. Ce processus est souvent très coûteux car il prend du temps et n'est pas sans risque d'erreur [50].

Supposons que nous voulions ajouter un état "A envoyer" à un objet stocké en base de données. Dans la table correspondante nous voyons qu'il y a une colonne "status". Celle-ci peut contenir "A générer", "Générer", "Envoyer", "Réponse OK" et "Réponse KO". Nous nous disons super ! Nous apportons notre modification en ajoutant "A envoyer". Nous nous apercevons, dés lors, que l'objet ne se traite plus correctement. Celui-ci engendrant de grosses erreurs, il nous force à retirer notre modification. Nous nous rendons-compte alors que le statut n'est pas uniquement géré par cette colonne "status" mais aussi par une autre "error". Cette colonne permet d'indiquer si l'objet a été généré avec une erreur ou non. Comme notre nouveau statut est inclut dans la colonne "status" et puisque nous n'avons pas vu qu'il était possible de "mal" générer l'objet alors, dans ces cas là, nous avions une erreur.

Nous pouvons nous rendre compte de deux problèmes avec cet exemple.
— Le premier est la non connaissance de cette colonne "error" et de son fonctionnement.
— Le deuxième est que d'un côté, nous avons une colonne pour gérer des erreurs pour un statut en particulier et d'un autre côté, nous avons dans la colonne contenant les statuts, deux états pour gérer les réponses en erreur ou non ("Réponse OK" et "Réponse KO").

Cet exemple est une simplification d'un cas qui nous est arrivé où il a fallu plus d'une semaine pour comprendre le fonctionnement des états et trouver une solution à une problématique simple à la base.

C'est pour répondre à ce genre de difficultés que **Meurice** [50] a, dans sa thèse, mis en avant une analyse statique permettant de localiser et lier automatiquement les requêtes créées dynamiquement via JDBC, Hibernate et JPA, et le schéma de base de données. Cette analyse se fait en trois temps.

1. Détecter tous les accès à la base de données.
2. Analyser chaque accès selon sa nature, JDBC, Hibernate ou JPA.
3. Regrouper les liens ainsi formés entre la base de données et le code. Il a constaté plusieurs limitations à cette approche liées à sa nature statique auxquelles il répondra en la combinant à de l'analyse dynamique.

Le tout est mis en 2D et en 3D avec l'utilisation de DAHLIA (DAtabase ScHema EvoLutIon Analysis). DAHLIA est un outil de visualisation interactive pour l'évolution du schéma de la base de données.

Il met aussi en évidence que les principes utilisés peuvent être étendus à d'autres types d'accès de bases de données ainsi qu'à d'autres langages de programmation.

Avant les recherches de **Meurice** pour sa thèse, **Nagy, Cleve et lui** [53] ont utilisé l'analyse statique pour identifier les queries et le code impacté par une modification du schéma de la base de données. Ils l'ont utilisée afin de simuler un changement sur le schéma (sous-section 2.2.2 à la page 21).

Afin de comprendre l'interaction entre l'application web et la base de données **Alalfi et al.** [21] proposent une nouvelle approche WAFA (Web Application Fine-grained Analysis). Cette approche consiste en l'extraction d'un modèle

avec une granularité fine. Cette extraction est faite via des analyses statique et dynamique du système. Pour la réaliser, ils font tourner l'application cible dans un environnement de test avec des cas de test. Ils enrichissent les requêtes afin de pouvoir les analyser par la suite. Les résultats sont regroupés dans une base de données pour être interprétés.

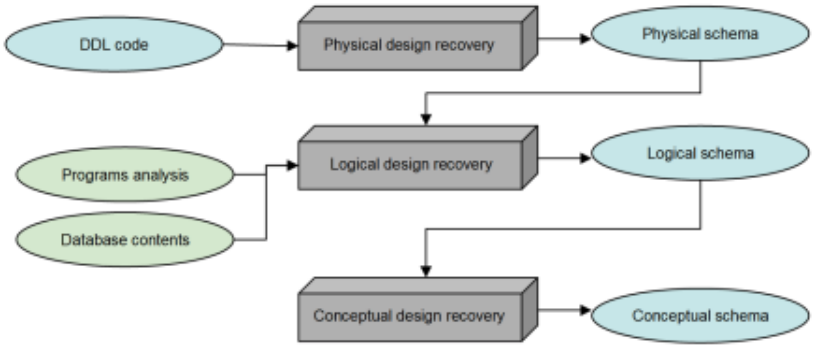


FIGURE 1.3 – Processus de rétro-ingénierie du design de la base de données [54]

Il existe aussi différents outils automatisant la création de modèles entité-relation sur base d'une base de données existante [35]. A l'origine, il en existait de nombreux. De nos jours, seuls quelques-uns subsistent tels que DB-main [3] et Visual Paradigm. Ils ont laissé la place à d'autres outils qui convertissent le schéma physique non plus en modèles entité/relation mais en diagrammes UML [35].

**Nesrine** [54] explique le principe de ces outils comme étant (Figure 1.3) :

1. **Récupération du design physique** (*Physical design recovery*) : Extraction du schéma physique sur base du code DDL de la base de données.
2. **Récupération du design logique** (*Logical design recovery*) : Raffinement du schéma physique en schéma logique avec l'aide d'informations supplémentaires et suppression d'éléments spécifiques au schéma physique (tels que les indexes).
3. **Récupération du design conceptuel** (*Conceptual design recovery*) : dérivation du schéma conceptuel sur base du schéma logique afin d'avoir un schéma le plus abstrait possible.

### 1.3.3 Le code

Avoir une bonne architecture et une base de données bien pensées, c'est bien, mais ça ne fait pas tout. Sans code, le développement d'applications est limité dans la diversité réalisable. Le code est au coeur de beaucoup d'applications. Celui-ci peut rapidement tourner au cauchemar pour les développeurs. **M. Feathers** [36] écrira en parlant du code "*Remember, code is your house, and you*

*have to live in it.*" [1]

En tant que développeur, on change du code pour vivre. Nous pouvons le modifier de deux manières. Une manière qui nous rend la vie plus simple et une autre qui la rend plus complexe. **M. Feathers** [36] nommera ces deux façons d'apporter un changement comme étant "*Edit and pray*" et "*Cover and modify*".

Il proposera plusieurs méthodes pour reprendre la connaissance sur une partie de code afin d'éviter de tomber dans l'"*Edit and pray*".

— Prendre des notes et faire un schéma. (Pas besoin de faire de l'UML. Quelques mots clés et des flèches entre eux permettent souvent de comprendre plus facilement le code).
— "Habiller" le code en apportant couleurs, lignes et dessins (implique de l'avoir imprimé) pour permettre une meilleure compréhension en regroupant les codes concernant un même sujet, aider à la compréhension de la structure du code ou encore, suivre un flux pour voir l'effet d'un changement.
— "*Scratch refactoring*" qui consiste à refactorer le code pour le comprendre. Casser la logique, changer les noms de variables et méthodes, simplifier le code pour comprendre ce que le code fait avant de réellement travailler dessus.
— Supprimer le code inutilisé. Supprimer tout le code qui n'est pas utile à votre compréhension.

Il existe aujourd'hui des outils permettant de faciliter ces méthodes. Les outils de versionning facilitent le retour en arrière en cas de mauvaises modifications. Ils permettent aussi d'essayer de nouvelles choses tout en laissant la possibilité de revenir à un état fonctionnel. La création de branches dans GIT est un bon exemple pour illustrer la possibilité d'essayer de nouvelles choses. Une simple commande et nous pouvons changer de branche et donc de version de code. Il est aussi très simple de supprimer une modification faite tout en laissant l'application dans un état stable.

Il existe aussi des outils d'aide au refactoring aidant le développeur dans son code. Certains IDEs intègrent même directement ces fonctionnalités de refactoring automatique. Intellij en est un exemple [8]. Il offre plusieurs fonctionnalités facilitant la vie au quotidien. Il marque les méthodes qui ne sont plus utilisées, permet d'extraire automatiquement du code sous forme de méthode ou encore propose la modification de code pour le rendre plus lisible.

La plupart des IDEs proposent d'ailleurs un debugger. Si le debugger sert souvent pour aider à comprendre comment se produit un bug, par sa nature, il peut aussi aider à comprendre comment fonctionne n'importe quelle partie de code. Il y en a même qui vont plus loin et permettent de revenir en arrière dans le debugging. Certains vont même jusqu'a proposer la modification à la volée [8]. Avec de telles fonctionnalités, il est envisageable de s'arrêter à un endroit du code et voir ce qui se passe si on modifie tels ou tels paramètres sans devoir lire tout le code ou écrire des tests faisant la même chose.

1. Michael Feathers. *Working Effectively with Legacy Code.* Prentice Hall Professional, 2004, page 58.

Avoir le diagramme de séquences de la fonctionnalité que nous voulons débugger faciliterait grandement le travail. Si il est possible de le refaire à la main en suivant le code, c'est un processus lent et fastidieux. Des outils tels que EasyCODE, Staruml, Visual-paradigm ou MoDisco [15, 17, 18, 26] permettent de générer des diagrammes UML. Si, avec ces outils, il est possible d'avoir une idée de l'architecture, il est aussi possible pour certains d'entre-eux de générer d'autres formes de diagrammes UML propices au parcours du code. Visual-paradigm ou MoDisco offrent la possibilité de générer un diagramme de séquence sur base du code.

Il existe aussi des outils tel que sonar [16] qui offre des métriques et des graphiques sur l'état du code. Si ces outils n'aident pas directement à comprendre le code, ils offrent la possibilité d'avoir une idée objective de l'état du code.

Enfin, même si ce n'est pas exactement le sujet de ce mémoire, il est bon de savoir qu'il existe des outils pour décompiler des logiciels [19], ce qui aide à leur compréhension si nous n'avons pas accès au code.

## 1.4 Analyse comparative

Depuis le début de la rétro-ingénierie, de gros progrès ont été apportés. Nous sommes passés d'algorithmes de traduction [35] à des outils permettant la modélisation du schéma de la base de données quelque soit son type [3, 35]. L'évolution entre les deux est notable. Nous sommes passés de "simples" algorithmes à des outils rendant possible la modélisation de n'importe quelle base de données relationnelle automatiquement.

Comme le montre le Tableau 1.1, il existe de nombreux outils pour aider à récupérer la connaissance d'une application [3, 8, 15, 16, 17, 18, 26, 50].
Ceux-ci couvrent en grande partie la question de la rétro-ingénierie. Certains vont aider à avoir une vue d'ensemble du système, par exemple, sous la forme de diagramme UML. Là où d'autres vont répondre à des questions plus précises comme la "mise en évidence des problèmes" par l'IDE. Cette mise en évidence aide, par exemple, à récupérer les méthodes non utilisées [8]. Chaque outil donne un point de vue différent sur la situation du logiciel.

Les différentes facettes de l'application sont fournies différemment d'une technique à une autre. Dans un cas, nous aurons des données brutes à traiter [21]. Dans un autre cas, nous aurons une visualisation graphique [50]. Cette dernière facilite grandement l'interprétation des données. Dans un autre cas encore, nous aurons des schémas permettant d'avoir une représentation structurée de la situation. Celle-ci est tantôt sous la forme de diagrammes UML, tantôt sous la forme de modèles entité-relation.

| **Auteurs ; Outils ; Techniques** | **Axe**[1] | **Méthodes** | **Sources** | **Résultat** | **Date** |
|---|---|---|---|---|---|
| EasyCode [15] ; StarUml [17] ; Visual-Paradigme [18] | Archi ; Code | Automatique | Code | UML | 2019 |
| MoDisco [26] | Archi ; Code | Automatique | Multi-Langue | UML | 2014 |
| IDE [8] | Archi ; Code | Automatique | Code | UML | 2019 |
| "*Telling the story of the system*" [36] | Archi | Manuelle | Tout | Phrase décrivant l'architecture | 2004 |
| Meurice [53] | BDD | Analyse statique | Queries | ? | 2016 |
| Meurice [50] | BDD | Analyse statique ; dynamique | Queries dynamiques | DAHLIA | 2017 |
| WAFA [21] | BDD | Analyse statique ; dynamique | Application | Base de données | 2009 |
| DB-main [3] ; Visual-Paradigme [18] | BDD | Analyse statique | Base de données | Entité-Relation | 2019 |
| Outils de modélisation UML [35] | BDD | Analyse statique | Base de données | UML | 2000 |
| Sonar [16] | Code | Automatique | Code | Etat du code ; Métriques ; Graphiques | 2019 |
| Notes et Schémas [36] | Code | Manuelle | Code | Schéma simple | 2004 |
| Habiller [36] | Code | Manuelle | Code | Code formaté ; Code colorié | 2004 |
| *Scratch refactoring* [36] | Code | Manuelle | Code | Code réarrangé | 2004 |
| Supprimer le code inutile [36] | Code | Manuelle | Code | Code nettoyé | 2004 |
| IDE - Habiller [8] | Code | Automatique | Code | Code formaté ; Code colorié | 2019 |
| IDE - Aide au refactoring [8] | Code | Automatique | Code | Code réarrangé | 2019 |
| IDE - Mise en évidence des problèmes [8] | Code | Automatique | Code | Code nettoyé | 2019 |
| IDE - Debugger [8] | Code | Automatique | Code | Simulation du comportement ; Parcours du code | 2019 |

[1] Les axes sont "Archi" pour architecture, "BDD" pour base de données et "Code".

TABLE 1.1 – Rétro-ingénierie, tableau récapitulatif

Avec l'arrivée des diagrammes UML, certains ont émis l'éventualité de la disparition progressive des modèles entité-relation pour la représentation du schéma de la base de données au profit de l'UML [35]. Cela fait près de 20 ans que cette supposition a été faite. Pourtant, nous sommes forcés de constater que les outils proposant la modélisation de la base de données ou la rétro-ingénierie de celle-ci vont privilégier le modèle entité-relation et non le diagramme de classes du langage UML même si l'outil propose l'utilisation de l'UML [18].

Dans notre pratique, nous avons pu observer que la modélisation ne se faisait plus au niveau de la base de données. Avec l'apparition des ORMs (*object-relational mapping*) et leur utilisation, nous nous concentrons plus sur la modélisation des classes permettant de faire le mapping avec la base de données. Nous n'avons ainsi qu'une seule manière de modéliser toute l'application et la possibilité de tout centraliser en un seul endroit.

Si ce constat s'avérait être généralisé à la majorité des projets, cela expliquerait l'absence de développement de la modélisation sous forme UML pour la rétro-ingénierie des bases de données.

Parmi tous les outils et techniques relevés dans ce chapitre, l'intégration de ceux-ci dans les IDEs est fort présente pour certains d'entre eux [8].

Cette intégration facilite des solutions manuelles telles que proposées par **Feathers** [36]. Il aborde, par ailleurs, le sujet. Il met toutefois en garde sur la fiabilité de ces outils. Si ceux-ci sont une aide précieuse permettant de gagner du temps, ils ne sont pas toujours fiables à 100%.

Nous soulignerons que dans notre pratique, nous utilisons au quotidien la plupart de ces aides et nous ne nous imaginons plus faire sans. Les avoir à disposition directement dans l'IDE est important dans ce contexte, car cela facilite et encourage leur utilisation.

Avec toute cette panoplie d'outils, nous pourrions nous dire qu'il ne reste plus rien à faire. Pourtant, **Brunelière et al.** [26] diront du *"Model Driven Reverse Engineering"* qu'il reste encore plusieurs challenges.

— Limiter les pertes d'informations dues à l'hétérogénéité des systèmes *legacy* existants.
— Améliorer la compréhension des systèmes *legacy*. Faciliter la lisibilité des modèles générés aussi bien par un développeur qu'un utilisateur lambda.
— Gérer la scalabilité des systèmes. En général, les systèmes *legacy* génèrent de très grands modèles dus à leur taille.
— S'adapter et apporter des solutions spécifiques aux besoins. Beaucoup d'outils *"Model Driven Reverse Engineering"* ciblent des techniques spécifiques à une technologie ou à un scénario de rétro-ingénierie.

De même, même si tous ces outils sont pratiques, ceux-ci ne savent récupérer ni la connaissance ni l'expérience des développeurs ou des utilisateurs afin de les mettre sous une forme compréhensible par un tiers. C'est là que les techniques manuelles gardent leur intérêt.

Parfois, comme "*Telling the story of the system*", il est difficile d'automatiser la technique. Avec des techniques d'intelligence artificielle nous pourrions peut-être arriver à quelque-chose, mais rien ne garantit la fiabilité. A part d'un point

de vue théorique, il est aussi nécessaire de se demander si un gain de temps est vraiment présent.

Aussi, dans le cas de "Notes et Schémas", l'utilisation d'un outil peut donner lieu à un surplus d'informations inutiles sur certains points et pas assez sur d'autres. La simple utilisation d'une feuille et d'un crayon peut largement être suffisant dans certains cas. De simples techniques manuelles peuvent suffire à notre problème du moment.

Nous pouvons dire qu'il n'existe pas de meilleure technique ou outil pour récupérer efficacement la connaissance d'un logiciel. Chaque technique apporte une vision différente à un même problème. Il faut pouvoir les utiliser au bon moment afin d'avoir toutes les clés pour la compréhension du système.

# Chapitre 2

# Analyse d'impacts

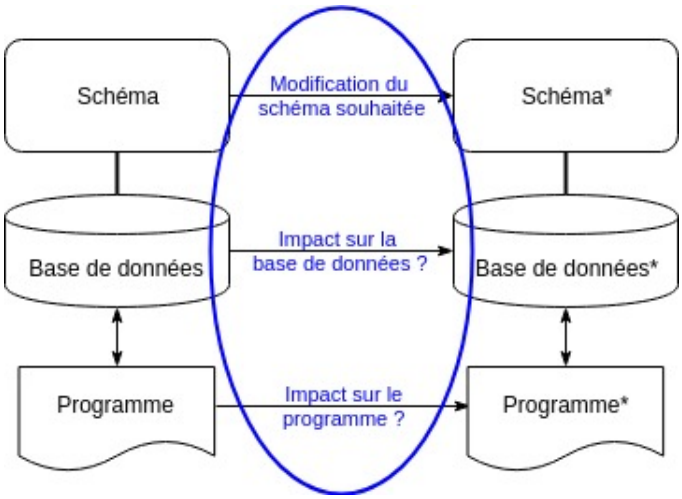


FIGURE 2.1 – évolution des logiciels, analyse d'impacts

Adapter l'application tout en gardant une consistance de celle-ci n'est pas chose aisée. Il faut identifier les lieux à modifier que ce soit dans le code ou dans la base de données puis comprendre comment apporter les changements pour évaluer le coût de ceux-ci pour déterminer quels sont les meilleurs choix [3, 10].

Dans le chapitre précédent, nous nous sommes concentré sur le système existant pour comprendre comment celui-ci fonctionnait. Dans ce chapitre, nous allons porter notre attention sur ce qu'il faut changer pour que le système ait toujours un comportement consistant (Voir Figure 2.1).

Dans notre pratique, nous avons pu constater un manque de conscience de l'importance de l'analyse d'impacts. C'est pourquoi, nous allons dans la section 2.1 expliquer l'utilité de l'analyse d'impacts. Par la suite, dans la section 2.2, nous recenserons au mieux les techniques existantes afin de pouvoir les comparer dans la section 2.3.

## 2.1 Pourquoi faire de l'analyse d'impacts ?

Faire l'analyse d'impacts à la main n'est pas une tâche simple, est peu fiable et est souvent très chère [52]. Elle demande souvent une grande connaissance de l'application, elle prend du temps et est risquée [50]. En effet, que ce soit par manque de temps, manque de connaissances ou manque de moyens, toutes les applications ne sont ni toujours correctement documentées ni toujours écrites selon les bonnes pratiques.

L'apparition de méthodes incrémentales, telles que les méthodes agiles, engendre une évolution continue de l'application et avec elle une demande de faire l'analyse beaucoup plus fréquemment sans pour autant diminuer la complexité de chaque incrément et donc les coûts de celle-ci [28].

Les changements sur la base de données ont un très gros impact sur le reste de l'application. Quand un changement apporté n'est pas approprié, cela peut engendrer des régressions sur le système d'origine [49]. Ces régressions peuvent être des exceptions qui apparaissent car une donnée est manquante ou est en trop voire encore, peuvent générer des erreurs dans des calculs. Les plus complexes à identifier et à corriger sont celles qui apparaissent uniquement avec des conditions tellement précises qu'elles n'arrivent que rarement ou encore en cas de traitement en parallèle difficilement reproductible. C'est pourquoi analyser les impacts est souvent nécessaire voire vital pour identifier les conséquences potentielles. Mais l'analyse d'impacts est loin d'être simple. Pour identifier les impacts dans le code depuis la base de données et vice-versa, il faut prendre en compte qu'aujourd'hui la plupart des applications utilisent des technologies qui construisent dynamiquement les queries au runtime, souvent via des ORM, ce qui complexifie l'identification des requêtes impactées par un changement [50, 53].

La plupart des applications évoluent en fonction des demandes tout au cours du temps. C'est encore plus vrai avec les méthodologies agiles. Ces changements réguliers entraînent une dégradation continue du code. C'est pour cette raison que beaucoup de recherches ont été faites sur des techniques de refactoring. [28] Le refactoring logiciel, selon **Chang**, a pour objectif de restructurer le code existant dans le but de modifier sa structure interne sans modifier son comportement externe. [28]

Ces dégradations sont aussi vraies sur la base de données et son schéma. Comme la fréquence des modifications du schéma de base de données augmente, une partie de la communauté des développeurs s'est tournée vers le NoSQL sans schéma qui offre une plus grande flexibilité par rapport aux bases de données relationnelles. Ce qui implique qu'il existe de plus en plus d'applications utilisant le NoSQL. [61] L'utilisation de ce type de bases de données est aussi à prendre en compte car le comportement n'est pas le même entre une base de données relationnelle et une base de données NoSQL. Dans l'exemple ci-dessous (inspiré de [61]), on veut faire évoluer un schéma comme suit :

Si, au niveau du code, l'impact sera équivalent que nous ayons une base de données relationnelle ou une base de données NoSQL, ce n'est pas le cas au

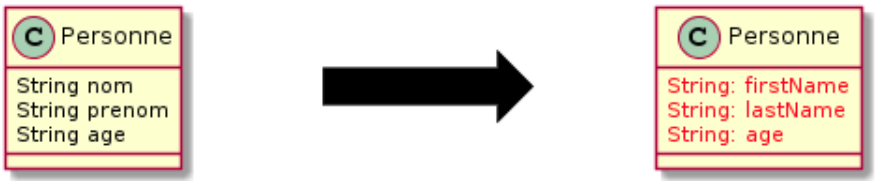


FIGURE 2.2 – Exemple de modification d'un schéma de base de données

niveau de la base de données. Si nous sommes sur une base de données relationnelle, nous devons changer les noms des colonnes. Alors que pour une base de données NoSQL, nous devons traiter les anciennes données, soit en les mettant toutes à jour en fonction du nouveau schéma, soit en chargeant les données sous les deux formats.

## 2.2 Approches existantes

L'objectif de l'analyse d'impacts est de limiter les risques d'un changement, que celui-ci vienne d'une nouvelle demande business ou d'un besoin de refactoring. Pour ce faire, il est important d'anticiper (sous-section 2.2.1) un maximum les impacts pour être en mesure, dans un second temps, d'identifier (sous-section 2.2.2) les endroits du code ou de la base de données impactés pour enfin proposer (sous-section 2.2.3) une solution au changement.

### 2.2.1 Anticiper

A la différence d'autres types de constructions faites en bois, métal, plastique ou autres qui s'usent avec le temps, le code, lui, ne se dégrade pas si on ne le modifie pas.[36] Le plus simple pour limiter tout risque serait donc de ne pas y toucher. Comme cela est rarement possible, il est donc important d'apporter des solutions à mettre en place lors de son écriture pour diminuer les risques ou tout du moins mettre des mécanismes en place pour aider à les identifier lors des futures modifications.

Pour pouvoir identifier les impacts, quelque soit la manière de faire, il faut connaître le fonctionnement de l'application. Plus celle-ci est grande et complexe, plus il devient difficile de s'y retrouver. Selon **L. Meurice** [50], le programme et la base de données sont rarement bien documentés. Avoir cette documentation complète peut aider à déterminer les risques et appréhender cette complexité. De même, des stratégies existent pour limiter la grandeur des applications tel que le découpage de celles-ci en modules ou encore en micro-services. A ce sujet **M. Fowler** [11] écrit que dans une application monolithique, il est compliqué de garder les modules correctement découplés et il privilégie l'utilisation de micro-services quand cela est possible pour faciliter l'encapsulation du code, facilitant la maintenance de celui-ci. Nous irons plus loin sur ce point quand nous aborderons le chapitre sur l'adaptation du programme (chapitre 4).

Afin de faciliter l'identification et de limiter les risques, **Grolinger et Capretz** [40] proposent, de la même manière qu'il existe des tests sur l'application, l'intégration dans ceux-ci des queries pour permettre d'aider le programmeur à identifier l'impact d'un quelconque changement au coeur de la structure de la base de données.

Modifier le schéma de la base de données ne se fait pas sans mal surtout quand celle-ci est fortement utilisée.C'est pourquoi **Papastefanatos, Vassiliadis, Simitsis et Vassiliou** [56] proposent l'utilisation des vues pouvant limiter l'impact de changements dans l'application mais en risquant de complexifier l'évolution de la base de données par la suite.

### 2.2.2 Identifier

Comme dit plus haut, l'analyse d'impacts n'est pas une tâche simple. Les applications étant de plus en plus complexes, il devient difficile d'identifier les impacts potentiels sur celles-ci. C'est pourquoi pas mal de recherches on été faites pour aider à l'identification des zones impactées par une modification via différentes stratégies et outils utilisant ces stratégies. L'objectif est de supporter les développeurs dans la maintenance de l'application, ainsi que de proposer des méthodes moins coûteuses que la manière manuelle.

Une manière d'aider à identifier les impacts est d'utiliser les graphes de dépendance. C'est ainsi que **Liu et al.** [49] proposent un nouveau graphe "attribute dependency graph" pour mettre en évidence les dépendances entre attributs de base d'une données et le programme via une programme d'analyse statique inter-procédural. En utilisant également les graphes de dépendances **Gardikiotis et Malevris** [38] apportent une solution pour aussi bien identifier les impacts dans le code source que dans les tests. Ils ont démontré via un outil implémentant leur solution, qu'il est possible d'aider à maintenir le logiciel suite à des modifications du schéma de bases de données.

Une autre approche consiste en l'utilisation de techniques d'analyses de programmes tels que "string analysis" pour voir où sont les impacts des changements de la base de données sur l'application. Sur base de celle-ci, **Maule et al.** [52] ont construit un algorithme plus performant qui permet d'identifier, via l'analyse de flux de données entre une application orientée objet et une base de données relationnelle, les impacts de changements du schéma de base de données sur l'application. L'algorithme consiste en la découpe en trois étapes pour arriver à identifier les impacts plus rapidement et plus efficacement. Ces étapes sont :

1. découpage du programme pour garder les parties impactées par une modification (retirer les parties inutiles).
2. analyse des flux
3. calcul des impacts. Grâce à cet algorithme, on peut soumettre un hypothétique changement pour permettre d'identifier les impacts de celui-ci.

De même, **L. Meurice** [50], comme nous avons pu en parler dans la section 1.3.2 à la page 11, a utilisé une analyse statique enrichie avec d'autres approches telles que la mise en place d'analyse "What-if analysis" et l'utilisation d'analyse visuelle (DAHLIA).

La méthode qui semble être la plus représentée est l'analyse d'impacts "What-if analysis" qui consiste à analyser ce qui se passe quand on change une donnée d'entrée, que ce soit au niveau de la base de données ou que ce soit au niveau de l'application.

Illustrons ce principe par un exemple. Prenons une application qui permet d'afficher et de modifier un catalogue. Supposons que nous voulions savoir où est utilisé le prix des éléments du catalogue (c'est notre input) car nous voulons modifier son utilisation en ajoutant la notion de devises. Avec la méthode "What-if analysis", nous allons simuler la variation du prix des éléments pour identifier ce qui est impacté. De là, nous serons capable de déterminer tout ce qui devra être fait pour ajouter la notion de devises.

Le présent exemple est extrêmement simple et il est facile de réaliser manuellement l'analyse d'impacts. Qu'en est-il pour une application de millions de lignes de code avec un schéma de base de données complexe ?

C'est pour répondre à cette problématique que plusieurs outils ont été développés.

Sur ce principe, **Papastefanatos et al.** [56] ont créé leur propre outil. Celui-ci offre des graphes et des métriques pour aider à l'identification des impacts sur la base de données. Il se base sur l'un de leurs travaux précédents [55] où ils ont apporté une amélioration d'une version antérieure d'un graphe permettant de modéliser les tables relationnelles, vues, activités ETL, contraintes BDD et queries SQL. Cette méthode permet de prédire les changements des sources d'un DW (data warehouse). Ils ont, de même, proposé un framework sur lequel nous reviendrons au point suivant (sous-section 2.2.3).

S'inspirant entre autre de cet outil, **Meurice et al.** [53] ont réalisé un outil permettant de simuler des changements sur le schéma de base de données et automatiquement déterminer les groupes de code impacté. Il fournit également des recommandations sur ce qu'il faut changer pour diminuer le risque d'inconsistance. Celui-ci a été conçu pour des systèmes Java utilisant des frameworks d'accès aux bases de données tels que JDBC, Hibernate et JPA. Dans sa thèse qu'il réalisa par la suite, **L. Meurice** [50], comme cité plus haut, alla plus loin en combinant la méthode "What-if analysis" avec d'autres méthodes pour arriver à de meilleurs résultats.

Quand une modification est apportée à une base de données relationnelle, elle est valable pour toutes les entités de celle-ci. Ce qui n'est pas le cas pour les bases de données NoSQL comme nous avons pu le voir dans l'introduction de ce chapitre. C'est pour cette raison que **Scherzinger, Cerqueus et Almeida** [61] proposent Controvol, un plugin à ajouter à un IDE.

Celui-ci se base sur l'historique des modifications des mappeurs pour identifier les incohérences entre ceux-ci et la base de données. De là, il peut même proposer certaines solutions que nous verrons dans la section suivante (sous-section 2.2.3).

De leur côté, **Chang et al.** [28] ont élaboré un cadre formel (un framework) pour faire évoluer la base de données et les queries avec comme objectif de

découvrir et résoudre les inconsistances de la base de données. Ils utilisent la logique épistémique pour faciliter l'identification des impacts.

Lors de l'élaboration de ces stratégies, qui se basent sur ce qui se passe dans le code quand on change le schéma de la base de données, simplement faire l'analyse sans réduire le champs de recherche des impacts possibles n'était pas efficace car il y a de ce fait plus de fichiers à traiter. Par exemple, modifier la base de données ne va pas impacter les fichiers de configuration de l'application, il n'est donc pas nécessaire de s'encombrer de ceux-ci quand on analyse les impacts d'une modification. C'est pour quoi, **Maule et al.** [52] proposent, dans leur algorithme permettant l'identification des impacts apportés par une modification, une découpe du programme pour ne garder que les parties pouvant être impactées par une modification (retirer les parties inutiles). Ce qui facilite la suite des traitements en les rendant plus rapides. Par ailleur, la plupart des outils permettant d'identifier automatiquement les endroits du code impactés par une modification utilisent des techniques de découpe du code afin d'aller plus vite.

### 2.2.3 Proposer

Localiser les endroits impactés par une éventuelle modification est une étape importante pour éviter des erreurs lors de l'évolution de la base de données et du code. Encore faut-il apporter la bonne modification. Si, dans les chapitres suivants, nous aborderons plus en détail sur comment changer la base de données et le code, il est intéressant de constater, comme nous avons pu le voir dans la section précédente, que certaines méthodes et outils pour identifier les impacts proposent également des changements dans l'application pour limiter de futurs impacts, voire d'anticiper certains changements.

Comme **Papastefanatos et al.** [55] qui, avec leur amélioration d'un graphe avec l'introduction d'un framework pour annoter les noeuds de celui-ci pour expliciter le comportement en cas de changement, offrent des règles pour savoir ce qu'il faut faire au coeur de la base de données suite à une mise à jour du schéma de celle-ci.

Si **Papastefanatos et al.** proposent des règles pour la base de données, **L. Meurice** [50] quand à lui va plus loin en recommandant des modifications dans le code. Suite à la localisation du code éventuellement impacté par une modification, il a été capable de fournir des recommandations pour le développeur pour éviter toute inconsistance.

De la même manière, **Scherzinger et al.** [61], en observant l'historique des modifications des mappeurs dans le cas de bases de données NoSQL, proposent les modifications des mappeurs pour éviter les inconsistances. Ils vont par exemple, suite à une modification de nom d'attribut d'un mappeur, proposer le code pour toujours charger les données sous l'ancien format.

## 2.3 Analyse comparative

Analyser l'impact d'un changement dans l'application est le coeur de toute modification. Si nous développons une application d'achat/vente en ligne qui possède un système de profil qui regroupe plusieurs informations comme le nom et le prénom ainsi que l'âge de la personne, que ces informations sont reprises un peu partout sur le site, comment cacher l'âge de l'utilisateur aux autres personnes allant sur le site ? Pour ce faire, il est nécessaire de connaître tous les endroits où l'âge est affiché. Et quand nous les connaissons tous, il est possible d'apporter les modifications aux bons endroits. Si nous n'identifions pas tous les endroits où l'âge est publique, il n'est pas possible d'apporter le changement.

Si, avec le rétro-Ingénierie, nous sommes capables de dire comment se comporte l'application, lors de ce chapitre, nous avons pu constater que nous étions capables d'identifier les impacts au niveau du code et de la base de données de manière relativement certaine et d'arriver jusqu'à des propositions de modifications pour éviter les incohérences.

Au cours du temps, les méthodes permettant d'analyser les impacts et les outils se basant sur ces principes se sont multipliées et améliorées. De nombreuses technologies ont évolué demandant de repenser la manière de faire évoluer l'application et avec elles de nouvelles manières d'analyser les impacts. L'utilisation de plus en plus conséquente du NoSQL a eu comme résultat la proposition de **Scherzinger et al.** [61] permettant l'adaptation des mappeurs. D'un autre côté, nous avons des méthodes d'analyse d'impacts renforcées au fur et à mesure que la recherche avance [50, 56]. Les recherches récentes laissent des portes ouvertes vers de nouvelles améliorations des méthodes, l'élargissement de celles-ci à d'autres langages.

L'un des problèmes soulevé dans plusieurs articles est la grandeur du code. Plus celle-ci est grande et complexe, plus il devient difficile de s'y retrouver. Pour résoudre ce problème on en vient à découper l'application en de plus petites parties pour faciliter l'analyse. [52] De là, il semble évident qu'une application bien découpée avec des modules bien séparés sera donc plus simple à analyser. Pour aller encore plus loin et s'assurer que même par la suite les modules continuerons d'évoluer indépendamment les uns des autres, nous pouvons envisager la mise en place de micro-services qui enlève le risque de ne pas respecter la découpe modulaire de l'application. [11]

Dans le cas de modifications liées au "refactoring" d'une partie de l'application, aussi grande soit-elle, il est important de s'assurer que nous ayons le même comportement avant et après. Pour ce faire, il existe de nombreuses manières d'y arriver. [36] Cependant, il est très bien de couvrir une partie de tests. Si nous n'avons pas fait une analyse d'impacts sur la partie que l'on veut changer, comment pouvons nous être sûrs que l'on n'oublie aucun impact ?

| **Auteurs** | **A**[1] | **I**[2] | **P**[3] | **Méthodes** | **Agit sur** |
|---|---|---|---|---|---|
| Grolinger et Capretz [40] | • | | | Unit test | BDD |
| M. Fowler [11] | • | | | Découpage du programme | BDD<br>Code |
| Papastefanatos et al. [56] | • | • | | What-if analysis | BDD |
| Liu et al. [49] | | • | | Graphe de dépendance | / |
| Maule et al. [52] | | • | | Découpage du programme<br>String analysis | BDD<br>Code |
| Gardikiotis et Malevris [38] | | • | | Découpage du programme<br>Test | BDD<br>Code<br>Test |
| Chang et al. [28] | | • | | Epistemic logic | BDD<br>Code |
| Meurice et al. [53] | | • | | What-if analysis | BDD<br>Code |
| Scherzinger et al. [61] | | • | • | Historique des modifications | BDD<br>(NoSQL)<br>Code |
| Papastefanatos et al. [55] | | • | • | What-if analysis | Data warehouse |
| L. Meurice [50] | | • | • | What-if analysis<br>Historique des modifications<br>analyse statique de programme | BDD<br>Code<br>(DISS) |

[1] Anticiper
[2] Identifier
[3] Proposer

TABLE 2.1 – Analyse d'impacts, tableau récapitulatif

Si on veut que l'application dure dans le temps et puisse être maintenue, il semble nécessaire de mettre en place, dès le départ, des mécanismes pour limiter les impacts des changements et donc de coder de manière propre [51] et de se remettre en question pour faire évoluer le tout harmonieusement.

Lors de notre recherche pour ce chapitre, nous avons constaté qu'il existait beaucoup d'articles scientifiques sur le sujet mais peu voir pas d'outils utilisés par les sociétés. Cette absence semble étrange vu l'importance qu'un effet de bord peut avoir si celui-ci n'est pas repéré. Nous supposons que cette absence est due au fait que, plutôt que d'utiliser des outils externes, les entreprises préfèrent utiliser les tests pour identifier les impacts, comme toute application devrait de toute façon être couverte par des tests [51]. Cette supposition reste à confirmer.

# Chapitre 3

# Changement de la base de données

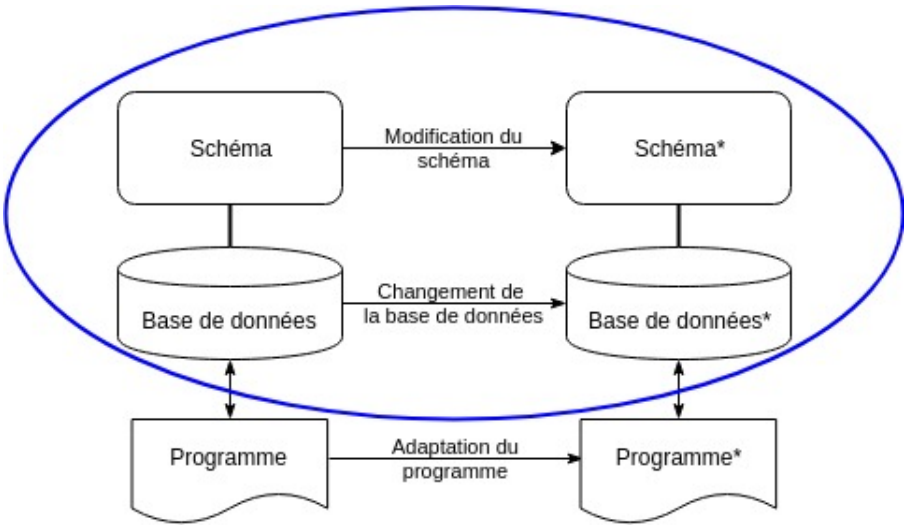


FIGURE 3.1 – Évolution des logiciels, changement de la base de données

Jusque là, nous avons vu comment reprendre la connaissance de l'application et de sa base de données ainsi que les impacts possibles d'un changement. Avec ces informations en main, nous allons voir concrètement comment apporter le changement.

Pour ce faire, nous allons voir quels types de bases de données sont utilisés ainsi que leurs différences (section 3.1). Pour chaque type de bases de données se comportant de manière différente face au changement, nous parcourerons ce qui existe pour apporter le changement à travers deux aspects. Le premier consiste en l'évolution en cours de développement de l'application (section 3.2). Nous le ferons à travers les deux types de bases de données utilisées aujourd'hui. Le second consiste en la migration de la base de données en une autre (section 3.3).

Enfin, nous ferons une rapide comparaison de ce dont nous aurons parlé dans les chapitres précédents (section 3.4).

## 3.1 Les types de bases de données

> "*But through all this time one thing has stayed constant—relational databases store the data.*" [1]

Depuis plusieurs années, les bases de données qui dominent sont les bases de données relationnelles. Comme l'expliquent **Sadalage et Fowler** [60], la question sur le stockage n'était pas quel type de bases de données nous allions utiliser mais plutôt quelle base de données relationnelle nous allions utiliser. Ce succès est dû entre autres au support qu'il y a autour de ce type de bases de données [23].

Depuis que la question du stockage des données est présente, plusieurs tentatives ont été menées pour résoudre ce problème. Nous sommes passés des fichiers plats, par les bases de données hiérarchiques et réseaux pour arriver aux bases de données relationnelles.

A partir de ce moment là, il y a bien eu différentes tentatives pour apporter d'autres types de bases de données telles que les bases de données orientées objet mais sans succès jusqu'à l'arrivée des bases de données NoSQL (il est plus approprié de les appeler non-relationnelles).

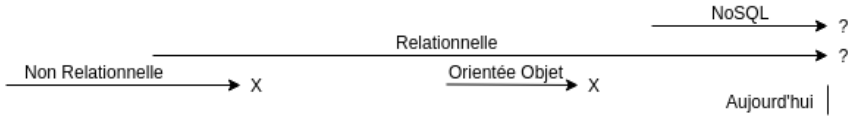


FIGURE 3.2 – évolution des types de bases de données au cours du temps

Les bases de données deviennent de plus en plus grosses et traitent de plus en plus de données. Suite à la transformation digitale des entreprises, celles-ci se dirigent vers des problématiques du big data. Les bases de données relationnelles répondent difficilement à ces problèmes [20].

Les sites web sont pour certains un exemple du manque de souplesse des bases de données relationnelles. Ils ont généralement besoin d'une structure flexible et évolutive. La plupart des sites web ne font que des opérations simples et ils n'ont nullement besoin de support d'opérations complexes [23].

Le NoSQL est là pour répondre à la demande de systèmes de données plus flexibles avec des opérations simples. Le NoSQL veut pouvoir répondre aux problèmes du relationnel, dans des situations telles que décrites ci-dessus, qui sont [23] :

— Performance inadéquate (exemple, pour du streaming)
— Structure des données trop rigide

1. Pramod Sadalage and Martin Fowler. Nosql distilled: A brief guide to the emerging world of polyglot persistence. 2012, page xiii.

— Certaines possibilités telles que la création de transactions et de queries complexes ne sont pas utiles (des opérations simples suffisent)
— La garantie de la consistance n'est pas toujours nécessaire. (Pour le bien de l'efficacité)

Aujourd'hui, un grand nombre d'organisations vont vers le NoSQL. Certaines envisagent les bases de données NoSQL pour de futurs développements tandis que d'autres envisagent de migrer leurs systèmes actuels vers ce type de stockage de données.

Apporter un changement dans la base de données ne se fait pas par simple envie du développeur. Dans la vie d'une application, les raisons pour changer la base de données sont les suivantes [36] :
— Ajouter une fonctionnalité
— Corriger un bug
— Améliorer l'architecture (le design)
— Optimiser l'utilisation des ressources

Tous ces changements engendrent inévitablement des problèmes de conception au cours du temps. Si ils ne sont pas appropriés, cela peut engendrer des régressions sur le système d'origine [49].

C'est pourquoi **Ambler et Sadalage** [22] à l'instar du "*Code smell*" introduit par **Fowler**, ont introduit le "*database smell*". Le principe est le même, ils ont identifié un catalogue de problèmes indiquant qu'il est probablement temps de faire un peu de refactoring. Ces indicateurs sont :
— Colonne pour plusieurs utilisations
— Table pour plusieurs utilisations (type d'entité)
— Données redondantes
— Table avec trop de colonnes
— Table avec trop d'entités
— "Smart" colonne (exemple : XML comme valeur)
— Peur du changement

Le refactoring de la base de données est plus complexe que celui sur le code. Là où, pour refactorer le code, il faut garantir le fonctionnement, pour la base de données, il faut en plus garantir la consistance des données [22]. Cela est d'autant plus complexe quand plusieurs applications utilisent une même base de données.

Apporter un changement dans la base de données ne se fait pas de la même manière que nous travaillions avec une base de données relationnelle ou une base de données NoSQL. Là où les bases de données relationnelles ont un schéma, les bases de données NoSQL en sont dépourvues. Cette différence fait que nous ne traitons pas un changement de la même manière.

D'un côté, nous devons modifier le schéma avant de migrer les données tandis que de l'autre, nous ne devons pas changer le schéma. Par contre, avant de migrer les données, nous devons au moins comprendre l'état dans lequel se trouve la base de données. Pour retrouver le schéma, il faut regarder les données (voir chapitre 1).

Il est évident que même en utilisant une base de données relationnelle, nous ne connaissons pas forcément le schéma. Mais dans ce cas là, nous ne savons pas non plus où apporter la modification.

| **Etape** | **Relationnelle** | **NoSQL** |
|---|---|---|
| 1 | Changer le schéma | Retrouver le schéma |
| 2 | Migrer les données | |

TABLE 3.1 – Étapes d'un changement sur une base de données

Apporter un changement dans la base de données, comme tout changement, peut être fait manuellement. A l'instar d'autres types de changements, il y a un haut risque d'erreurs en agisant ainsi [24]. C'est la raison pour laquelle il existe des solutions pour aider la migration [33].

## 3.2 Evolution continue de la base de données

Il est connu des chercheurs et des développeurs qu'apporter une modification au niveau de la base de données impacte de manière conséquente les données et les queries. Ces modifications peuvent engendrer des problèmes d'intégrité dans les données. Elles demandent la maintenance des queries causant des arrêts du système [33]. La mise à jour des queries est un travail coûteux et risqué surtout si il est réalisé manuellement.

Cette problématique est d'autant plus importante pour les sites web qui demandent d'être disponibles sans interruption. Nous pouvons voir aussi que les entreprises ne veulent plus de temps d'arrêt dans la mise en production [33].
Nous avons besoin de trouver des solutions pour [33]
— Minimiser le temps d'arrêt du système.
— Diminuer le risque d'erreurs lors de la migration.
— Avoir l'historique des modifications.
— Avoir la possibilité de revenir au schéma précédent.
— Avoir des études de cas pour résoudre de futurs problèmes.

Comme nous en avons parlé plus haut, l'évolution de la base de données ne se fait pas de la même manière selon le type de bases de données que nous avons. Dans la suite de cette section, nous expliquerons comment faire évoluer les bases de données relationnelles dans la sous-section 3.2.1 ainsi que les bases de données NoSQL dans la sous-section 3.2.2. Nous ne parlerons que de ces deux types de bases de données car ce sont les plus représentées sur le marché actuel. Nous ne parlerons que d'évolution au cœur d'une même plateforme dans cette section.

### 3.2.1 Les bases de données relationnelles

**Rahm et Bernstein** [59] ont construit une bibliographie en ligne relative aux publications sur l'évolution des schémas. Ils ont catégorisé les publications selon plusieurs dimensions.

Une approche proposée par **Papastefanatos et al.** [55] telle que vue dans le chapitre 2 offre une aide dans la migration des données. Leur amélioration du graphe offre des règles pour savoir ce qu'il faut faire au cœur de la base de données suite à une mise à jour du schéma.

**Curino et al.** [33] ont développé, pars la suite, un outil PRISM++, ainsi que des techniques pour faire évoluer automatiquement la base de données. Avec leur solution, ils veulent minimiser le temps d'arrêt de l'application dû à la mise à jour de la base de données. L'outil supporte, en plus de ce que propose la version précédente PRISM, l'évolution des contraintes d'intégrité et modifie automatiquement les queries et update suite à une modification du schéma de la base de données.

Leur travail s'est principalement concentré sur la difficulté de gérer les updates et les queries legacy. Pour ce faire, ils fournissent un ensemble de requêtes SQL basé sur la sémantique SMO ("*Schema Modification Operators*" [14]). Ils ont mis à jour la sémantique SMO pour pouvoir supporter les updates. Pour ce faire, ils ont intégré les opérations de modification des contraintes d'intégrité.

Ils proposent une solution pour migrer les données de la version x vers la version x+1 mais aussi pour pouvoir revenir à la version x.

**Ambler et Sadalage** [22] ont définit six catégories principales dans le cas d'opérations d'évolution d'une base de données : transformation, structure refactoring, integrity refactoring, architecture refactoring, data quality refactoring et method refactoring. **Curino et al.** se sont inspiré de ces opérations pour la réalisation de PRISM++.

De même, **Qui** [57] la complète en une liste exhaustive de 24 opérations de changement du schéma.

Sur le marché, il est possible de trouver des outils pour automatiser partiellement ou totalement la migration de la base de données.

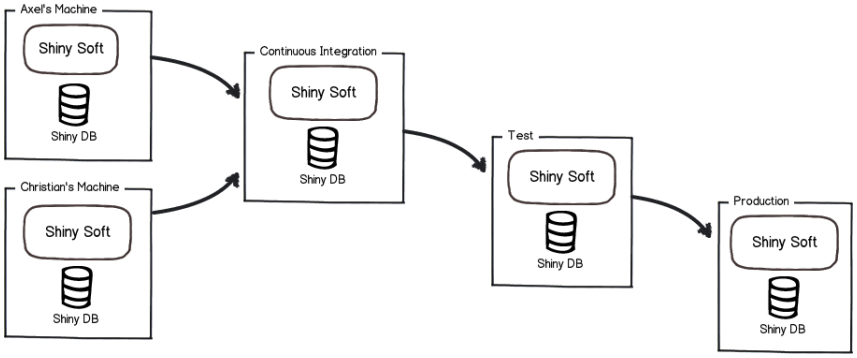


FIGURE 3.3 – Migration avec un outil de version de contrôle de la base de données [5]

Parmi ceux-ci, nous avons les outils de version de contrôle de la base de données. **Liquibase** [9] et **FlywayDB** [5] en sont des exemples. Leur fonctionnement est simple, vous écrivez les modifications que vous voulez apporter à la base de données. Une fois les modifications mises en place, celles-ci sont automatiquement appliquées sur chaque environnement déployé comme représenté sur la Figure 3.3.

Ces outils permettent non seulement d'appliquer un changement d'une version x vers une version x+1 mais également de revenir en arrière. Même si en pratique, le retour en arrière n'est pas toujours évident. Surtout si les changements sont "destructeurs" tels que un drop, un delete, un truncate, ... [5]

Nous avons aussi les ORMs tels que **Doctrine** en PHP [4] ou encore **Hibernate** en Java [6]. Ceux-ci permettent d'apporter des modifications dans le schéma de la base de données tout en faisant le lien entre le code et les queries (voir chapitre 4. Il est même possible de combiner liquibase et hibernate [10].

Avec les outils décrits ci-dessus, nous avons des réponses pour un bon nombre des problématiques au changement de la base de données. Nous avons diminuer le risque d'erreurs en automatisant la migration et en la rendant identique quelque soit l'environnement. Nous pouvons à tout moment voir l'historique des modifications que ce soit en direct avec les outils de version de contrôle de la base de données ou encore en utilisant un ORM combiné à un outil de gestion de version du code. Nous pouvons aussi revenir en arrière. Ne reste plus que le temps de migration.

Nous pouvons diminuer le temps d'arrêt de la base de données et du logiciel pour qu'il devienne insignifiant. **Blue-Green Deployment** est une solution proposée par **Fowler** [1] dont nous parlerons plus en détail dans le chapitre suivant. C'est une solution pour déployer une nouvelle version avec possibilité de revenir rapidement en arrière en cas de problème.

### 3.2.2 les bases de données NoSQL

Il existe plus d'une centaine de systèmes NoSQL [2] avec chacun son interface propre [23]. Ceci rend plus complexe l'évolution de ces systèmes.

De plus, il existe quatre groupes de bases de données NoSQL décrits par **Hecht et Jablonski** [43]. Ceux-ci sont : "key value", "document", "column" et "graph". Chaque groupe ayant sa propre manière de fonctionner. Avoir des techniques ou des outils traitant l'ensemble n'est pas simple [20].

De la même manière qu'il existe des outils tels que DB-main pour les bases de données relationnelles, **Abdelhédi et al.** [20] proposent des règles pour passer d'un schéma conceptuel vers un schéma d'implémentation pour une plateforme NoSQL orientée colonne. Afin de limiter les impacts de l'évolution des technologies des plateformes NoSQL, ils introduisent un schéma logique. Leur approche se base sur le "*Model Driven Architecture*".

D'autres travaux existent sur le sujet mais peu selon eux. Leurs objectifs est de poursuivre vers des systèmes NoSQL orientés graphes.

2. `https://db-engines.com/en/ranking_categories`

Dans le même domaine de recherche, nous avons **Atzeni et al.** [23] qui veulent standardiser les interfaces. Pour réaliser cette standardisation, ils introduisent un langage se voulant commun pour tous les systèmes NoSQL, SOS dont nous traiterons dans le chapitre suivant sur la co-évolution du code et de la base de données.

La recherche sur la manière de faire évoluer une base de données NoSQL étant assez récente, il existe peu de travaux sur le sujet.

Toutefois, nous avons déjà pu parler d'un outil en particulier dans le chapitre 2 (Analyse d'impacts). Controvol est un plugin développé par **Scherzinger et al.** [61]. En observant l'historique des modifications des mappeurs dans le cas de bases de données NoSQL, Controval propose les modifications des mappeurs pour éviter les inconsistances.

## 3.3 La migration

Jusqu'à présent, nous n'avons parlé que de l'évolution au cœur d'une base de données en particulier et des solutions pour la changer. Il existe aussi des techniques de migration. L'objectif de la migration est d'avoir une situation équivalente dans la nouvelle technologie [24]. Par exemple, si nous migrons une base de données Oracle vers une base de données MySQL, nous voulons que le schéma et les données restent les mêmes d'un côté comme de l'autre.

Migrer une base de données est un processus qui coûte cher et qui est complexe. Pourtant, il est parfois nécessaire de le faire pour pouvoir satisfaire de nouvelles demandes business [30]. Ce qui nous amène à deux problématiques majeures à résoudre [30] :

— La conversion vers le nouveau paradigme.
— L'adaptation du programme.

Comme expliqué au début de ce chapitre, pendant une assez longue période, seules les bases de données relationnelles étaient utilisées. Cette utilisation a fait que la plupart des articles sur la migration parlaient de migrer vers une base de données relationnelle. Nous n'avons trouvé que quelques exceptions avec les bases de données orientées objet. Ce n'est qu'avec l'apparition et la popularisation du NoSQL que nous avons vu l'arrivée en masse sur la migration vers ce type de bases de données.

Nous allons suivre cette logique et découper cette section suivant celle-ci. Nous parlerons d'abord des migrations vers les types de bases de données relationnelles et orientées objet (sous-section 3.3.1). Ensuite, nous nous pencherons sur les migrations du relationnel vers le NoSQL (sous-section 3.3.2.

### 3.3.1 Migration vers le relationnel et l'orienté objet

**Lin** [48] et **Cleve** [30] nous parlent de techniques ainsi que d'outils pour migrer de systèmes non-relationnels tels que les fichiers plats, les bases de données hiérarchiques ou encore les bases de données réseau.

La migration peut aussi se faire entre bases de données relationnelles. Pour cela, il existe plusieurs outils qui permettent de passer d'une base de données à une autre [2, 7].

Ce type de migration est plus simple car nous restons sur le même paradigme. De plus, avec l'utilisation des ORMs, il est même envisageable de ne pas à avoir besoin de modifier le code [4, 6].

D'un autre côté, **Behm et al.** [24] proposent une solution pour réaliser la migration d'une base de données relationnelle vers une base de données orientée objet. Ils génèrent un programme de migration. Celui-ci est généré sur base des transformations du schéma pour passer du schéma relationnel vers celui orienté objet. Une fois la migration faite, il reste nécessaire d'adapter le code pour modifier les requêtes pour correspondre à la nouvelle base de données et son nouveau schéma. Pour que ces modifications soient apportées automatiquement, il faut plus de travail et une meilleure compréhension.

Nous remarquerons qu'ils font la supposition que nous connaissons le schéma d'arrivée. Ils nous apprennent également que les paradigmes relationnels et orientés objet sont trop différents pour être migrés de l'un à l'autre automatiquement.

### 3.3.2 Migration du relationnel vers le NoSQL

Avant de migrer vers le NoSQL, **Zhao et al.** [65] se sont penchés sur la possibilité de le faire. Pour cela, ils ont comparé les bases de données relationnelles avec une NoSQL orientée document (plus exactement MongoDB).

Ils sont arrivés à la conclusion que la migration est réalisable car les deux types de bases de données ont les mêmes possibilités.

Une manière simple de migrer les données d'un type à l'autre est l'utilisation d'un *mapping one to one*. Nous avons notre base de données relationnelle avec son schéma et ses données. Nous traduisons ceux-ci dans la base de données NoSQL de notre choix.

Cette solution simple a ses inconvénients. Les deux types de bases de données ne se comportent pas exactement de la même manière. D'un côté, nous avons les bases de données relationnelles qui supportent jointures et transactions tandis que de l'autre côté, le NoSQL ne le supporte pas [60].

Dans le but de résoudre ce problème, nous avons plusieurs auteurs qui ont travaillé sur le sujet améliorant au fur et à mesure la technique.

**Ghaliza et al.** [39] ont d'ailleurs parcouru différentes recherches sur le sujet en en faisant un résumé.

Il y a plusieurs étapes pour migrer d'une base de données vers l'autre.

— Nous avons la construction d'un langage commun, dont nous avons déjà parlé dans la sous-section 3.2.2, qui nous offre une uniformisation de la migration.
— Pour migrer d'une base de données à une autre, il faut bien entendu migrer les données.

— D'autre part, les requêtes d'accès à la base de données doivent être adaptées.
— Enfin, Il est bon de savoir si la migration s'est bien passée.

Migrer les données comme vu ci-dessus n'est pas trivial. Il existe par ailleurs différentes approches.

Nous avons par exemple, **Liang et al.** [47] qui ont créé un modèle de transition. Toutefois, celui-ci doit être maintenu car il est au niveau physique.

**Rocha et al.** [34] sont partis sur un framework automatisant la construction de la structure en NoSQL orienté colonne. Leur framework, NoSQLLayer, en plus de construire la nouvelle structure, va également migrer les données.

Pour migrer les données, **Zaho et al.** [66] utilisent l'agrégation de ceux-ci pour éliminer les jointures. Le problème de ce système est la taille considérable de la base de données après migration.

C'est pourquoi, **Yoo et al.** [64] ont amélioré leur solution en utilisant la notion de "*Column-level denormalization*". Grâce à cette technique ils sont parvenus à diminuer grandement la taille de la base de données.

De même, **Lee et Zheng** [46] exploitent les principes de dénormalisation et de duplication pour migrer les données vers une base de données NoSQL orientée colonne.

De leur côté, **Stanescu et al.** [62] utilisent les relations entre les tables pour agréger les données. Le problème est que les relations ne sont pas toujours explicitement déclarées dans le schéma. Dans certain cas, ces relations sont implicites et utilisées dans l'application [31].

## 3.4 Analyse comparative

Avec le Tableau 3.2 reprenant les articles et outils dont nous venons de traiter, nous constatons plusieurs choses.

Tout d'abord, comme nous avons déjà pu en parler plus haut, il y a eu plusieurs types de bases de données qui se sont relayés. Le nouveau type de bases de données remplaçait le ou les anciens, si ce nouveau était plus adapté. Nous pouvons dire que, avant l'arrivée du NoSQL, il n'y avait de la place que pour les bases de données relationnelles et que toutes les autres tentatives ont été écartées. Nous pensons tout particulièrement aux bases de données orientées objet.

Nous sommes donc en droit de nous demander si le NoSQL ne va pas remplacer le relationnel. Relationnel et NoSQL ne traitent pas de la même chose. D'ailleurs, certaines sociétés se sont vues devoir revenir à des solutions relationnelles plus adaptées à leurs problématiques [32]. Il nous semble donc fort peu probable que le NoSQL remplace le relationnel (en tout cas pas d'ici les prochaines années).

| **Auteurs et outils** | **Méthodes** | **Type**[1] | **Concerne** | **Date** |
|---|---|---|---|---|
| Rahm et al. [59] | Résumé d'existant | Cont | Rel[2] | 2006 |
| Papastefanatos et al. [55] | Amélioration du graphe | Cont | Rel[2] | 2007 |
| Curino et al. [33] | Automatisation de l'évolution | Cont | Rel[2] | 2012 |
| Ambler et Sadalage [22] | Définition d'opérations de changement | Cont | Rel[2] | 2006 |
| Qiu et al. [57] | Définition d'opérations de changement | Cont | Rel[2] | 2013 |
| Flywaydb [5] ; Liquibase [9] | Outils de version de contrôle | Cont | Rel[2] | 2019 |
| Abdelhédi et al. [20] | Schéma conceptuel -> logique | Cont | NoSQL | 2016 |
| Atzeni et al. [23] | Interface standard | Cont | NoSQL | 2012 |
| Scherzinger et al. [61] | Support contre les inconsistances | Cont | NoSQL | 2015 |
| Lin [48] | Résumé d'existant | Mig | Non-Rel[3]–> Rel[2] | 2008 |
| Cleve [30] | Résumé d'existant | Mig | Non-Rel[3]–> Rel[2] | 2010 |
| Convert-in [2] ; Ispirer [7] | outils | Mig | Rel[2]–> Rel[2] | 2019 |
| Doctrine [4] ; Hibernate [6] | ORM ; ODM | Mig | BDD supporté –> BDD supporté | 2019 |
| Behm et al. [24] | technique | Mig | Rel[2]–> OO[4] | 1997 |
| Ghotiya et al. [39] | Résumé d'existant | Mig | Rel[2]–> NoSQL | 2017 |
| Liang et al. [47] | Modèle de transition | Mig | Rel[2]–> NoSQL | 2015 |
| Rocha et al. [34] | Migration de la structure et des données | Mig | Rel[2]–> NoSQL | 2015 |
| Zhao et al. [66] | Agrégation | Mig | Rel[2]–> NoSQL | 2014 |
| Stanescu et al. [62] | Agrégation | Mig | Rel[2]–> NoSQL | 2016 |
| Lee et Zheng [46] | *Column-level denormalization* | Mig | Rel[2]–> NoSQL | 2015 |
| Yoo et al. [64] | *Column-level denormalization* | Mig | Rel[2]–> NoSQL | 2018 |

[1] Les types sont "Cont" pour l'évolution continue et "Mig" pour la migration
[2] Rel = Relationnel
[3] Non-Rel = Non-Relationnel tels que les fichiers plats (pas NoSQL).
[4] OO = Base de données orientée objet

TABLE 3.2 – Changement de la base de données, tableau récapitulatif

Dans le Tableau 3.2, nous pouvons voir que les recherches sur le NoSQL sont récentes et pour cause, cela ne fait pas si longtemps que le NoSQL existe. Nous sommes encore qu'au début pour le NoSQL. Les outils pouvant supporter ce type de bases de données ne font qu'apparaître [4, 6].

Pour le relationnel, c'est une autre histoire. Les outils sont présents depuis bien longtemps [4, 5, 6, 9]. D'ailleurs, nous avons pu voir dans notre pratique que le relationnel est toujours très utilisé même pour des problématiques demandant du NoSQL par manque d'outils et donc par facilité.

Un élément marquant de ce chapitre est le grand nombre d'articles regroupant les connaissances sur la base de données, que se soit pour l'évolution continue ou pour la migration [30, 39, 48, 59]. Ce point ainsi que le fait que nous ayons facilement trouvé des articles parlant du sujet de ce chapitre nous montre qu'il y a un grand intérêt pour cet aspect de la recherche pour l'informatique.

# Chapitre 4

# Adaptation du programme

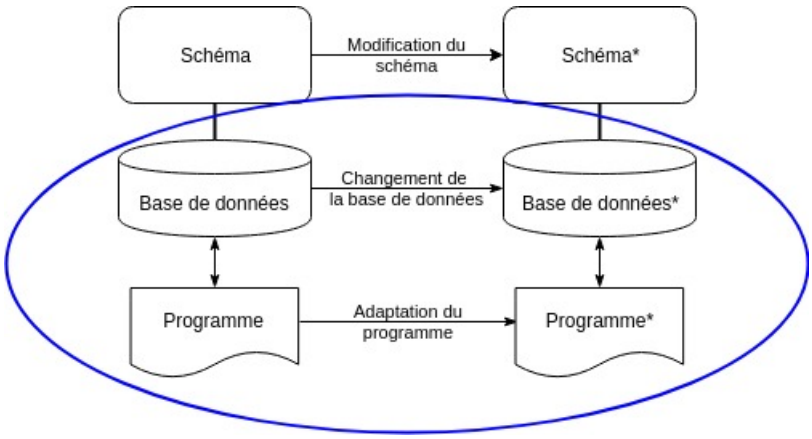


FIGURE 4.1 – évolution des logiciels, adaptation du programme

> "*Even the most disciplined development team, knowing the best principles, using the best patterns, and following the best practices will create messes from time to time.*" [1]

Les développeurs changent le code pour vivre. Il existe de multiples manières de le faire. Certaines rendant les futures modifications plus simples et d'autres les rendant plus compliquées.

Apporter les bonnes modifications n'est pas une tâche simple. Rien ne garantit qu'un choix judicieux à un moment ne devienne un mauvais choix à cause d'une nouvelle modification à mettre en place.

**Brooks et Martin** [51] développent et mettent en avant que nous devons revoir régulièrement des parties du logiciel.

Dans ce chapitre, nous nous focaliserons sur les modifications sur le programme ainsi que sur son évolution dans son ensemble (Figure 4.1).

1. Michael Feathers. *Working Effectively with Legacy Code.* Prentice Hall Professional, 2004, page XIV.

## 4.1 Le code et ses pratiques

Ce que veulent les entreprises, c'est fournir du logiciel. Ils veulent avoir le moins de bugs possible et ils veulent que l'application soit la plus facilement maintenable.

De plus, ils veulent avoir une solution qui soit rapidement mise en production [51].

Pour arriver à cette finalité, **R. Martin** [51] et **M. Feather** [36] donnent, tous les deux, une vision de l'évolution du code.

Dans cette section, nous expliquerons, sur base des points de vue de **R. Martin** et **M. Feather**, ce qu'est un code de qualité dans la sous-section 4.1.1. Nous mettrons tout particulièrement l'accent sur les tests dans la sous-section 4.1.2 et nous aborderons une réflexion sur ce que nous pouvons faire avec du code dit legacy dans la sous-section 4.1.3

### 4.1.1 La qualité du code

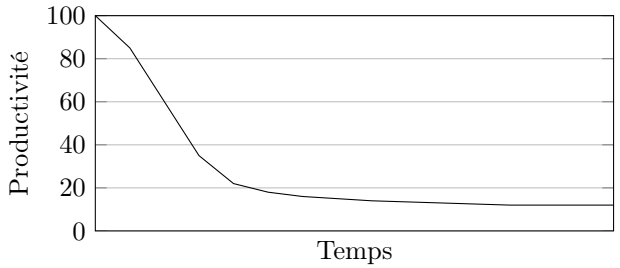


FIGURE 4.2 – La productivité au fil du temps [51].

Pour **R. Martin** [51], aller vite veut dire garder le code propre. Il met en avant que nous passons le plus clair de notre temps à lire du code. Si celui-ci n'est pas lisible, pas propre ou encore pas à l'endroit attendu alors, nous perdons notre temps. Nous pouvons lire dans son livre que "garder un code propre n'est pas une question de coût il s'agit d'une question de survie professionnelle." [2].

La Figure 4.2 illustre l'effet du mauvais code sur la productivité.

Le code n'est jamais parfait. Il dépend également de la qualité de l'équipe qui l'implémente [11]. C'est pour cela qu'il faut faire attention à sa construction, y compris à son architecture.

Pour répondre à ce besoin, les organisations utilisent le rôle d'architecte. Mais seul, celui-ci ne sait ni garantir la maintenance ni l'évolution correcte de l'architecture. C'est en travaillant en collaboration avec les développeurs qu'il est possible de garder une cohérence dans l'application.

2. R.C. Martin. *Coder proprement.* Référence : programmation. Pearson, 2013, page 5.

Quand les équipes ne sont pas conscientes de l'architecture de leur système, il tend à se dégrader car [36] :

— Le système est tellement complexe qu'il n'y a pas de vue d'ensemble.
— Le système est tellement complexe que cela prend longtemps pour avoir une vue d'ensemble.
— L'équipe est dans un mode d'urgence tel que la vue d'ensemble est perdue.

### 4.1.2 Les tests

J'ai eu l'occasion de voir qu'en ce qui concerne les tests, nous en parlons beaucoup mais nous en faisons peu ou pas correctement. Depuis que j'ai commencé à étudier l'informatique, les tests ont toujours été considérés comme l'une des choses les plus importantes. Par contre, lors de ces cours et, par la suite, dans les entreprises où j'ai travaillé, les tests n'étaient soit pas présents, soit de mauvaise qualité, soit ceux-ci prenaient tellement de temps à être exécutés que dans la vie du process de développeur, ils n'étaient pas pris en compte. Pourtant, tous mes collègues semblaient être d'accord sur le fait que les tests sont importants.

D'ailleurs, **M. Feathers** [36] dit du code que s'il n'y a pas de test qui le couvre alors, peu importe son implémentation, c'est du mauvais code. Nous ajouterons que les tests, en plus d'être présents, doivent être pertinents.

Qui n'a jamais vu un test qui n'était là que pour flatter l'algorithme de couverture de tests ? Un test qui n'affiche jamais d'erreur suite à une modification de la logique du code couvert n'est pas un bon test.

Les langages de programmation n'aident pas à corriger ce phénomène. En effet, ils supportent rarement bien l'écriture de tests [36].

Il existe toutefois de nombreux outils pour aider à l'écriture des tests. Les frameworks de tests unitaires ou d'intégration facilitent grandement leur écriture, compensant quelque peu l'imperméabilité des langages. L'utilisation de "mocks" et de "fake objects" aide également à l'écriture des tests en simulant le comportement d'objets inaccessibles ou en testant une partie de code spécifique [12]. Il faut cependant faire attention à leur utilisation. En effet, si tout le code est "mocké", rien n'est testé.

De plus, il faut trois jours avant d'avoir un feedback sur les résultats des tests, ceux-ci n'aident donc pas le développement au jour le jour. C'est la raison pour laquelle il faut que ces tests aillent vite. Chaque type de test est important mais il faut privilégier les tests les plus rapides [36].

Nous avons vu, par ailleurs, l'apparition de techniques favorisant l'écriture de tests. Le "test driver developpment" (TDD) est une technique où les tests sont écrits avant le code. Ce principe force à réfléchir autrement. Nous pouvons lire dans "working effectively with ..." [36] que les singletons ne sont pas "mockables" par nature, (une seule instance est autorisée).

**Guerra et Aniche** [42] nous mettent toutefois en garde sur son utilisation. Il ne faut pas croire que le fait d'écrire des tests en premier lieu nous protège de toute erreur. C'est pour cela qu'ils présentent une utilisation du TDD qui se veut efficace.

De même, **M. Feathers** [36] propose également le TDD mais va plus loin en soulignant que dans le cas de réécriture du code (refactoring), il est important d'avoir des tests. Ces tests, dans le cas de refactoring complet ou partiel, nous couvrent et servent à garantir la bonne modification.

### 4.1.3 Le code legacy

Dans la vie d'un logiciel, le code change. Nous pouvons faire tout ce que nous voulons, un jour ou l'autre, nous devons repasser dessus.

Nous pouvons supposer que tout ce que nous avons écrit précédemment dans ce chapitre a été appliqué mais rien n'est moins sûr.

Il existe beaucoup de livres et articles parlant du développement d'applications partant de zero et expliquant comment les réaliser correctement et simplement. Il est plus rare de trouver des références telles que "*Working Effectively with Legacy Code*" [36] et "*Coder proprement*" [51] traitant du code legacy.

Là où **R. Martin** [51] veut aider à identifier le bon code du mauvais code en donnant une idée de ce qu'est pour lui du code propre. *M. Feathers* [36], lui, accompagne le développeur pour l'aider à se débarrasser de ce mauvais code.

Pour cela, il met en avant l'existence d'outils d'aide au maintien du code.
— Des outils aidant au refactoring (parfois directement intégrés à l'IDE).
— Des librairies de mocks
— Des frameworks de tests unitaires
— Des frameworks de tests d'intégration
Si ces outils sont une bonne aide, ceux-ci seuls ne suffisent pas à rattraper toutes les dérives qu'elles soient dues à une mauvaise conception ou à un changement de comportement "normal" (un changement dans le business par exemple). Ces outils ne savent pas tout faire.

*M. Feathers* [36] propose plusieurs approches afin d'aider à la réécriture partielle ou complète du code complémentaire à ces outils. De manière plus large, il donne des aides à la programmation.

Les solutions qu'il fournit ne sont pas forcément idéales d'un point de vue architectural. Elles ont, malgré tout, l'avantage de permettre l'écriture de tests et donc de sécuriser et faciliter la réécriture ou modification de la partie concernée.

L'objectif de ces techniques ne sont pas de garder une situation où l'architecture est détériorée mais bien d'être une étape dans son remaniement.

## 4.2 La co-évolution

Beaucoup d'approches pour l'évolution de la base de données seule ont été menées avant de considérer la co-évolution de celle-ci avec l'application l'utilisant [28, 50]. Il existe peu de recherches qui traitent de ce sujet.

Nous avons les ORMs (*object-relational mapping*) tels que **Doctrine** en PHP [4] ou encore **Hibernate** en Java [6]. Ceux-ci facilitent grandement la vie des développeurs en facilitant le maping entre le code et la base de données. Ils facilitent la création de queries diminuant par la même occasion le besoin de les mettre à jour.

Ces outils vont même plus loin. Ils génèrent les modifications du schéma de la base de données via de simples commandes. Si cette génération n'est pas toujours parfaite [4], il n'en reste que les modifications n'en sont que plus rapides car automatisées.

Il est même possible de combiner liquibase et hibernate [10].

Dans le même domaine de recherche, nous avons **Atzeni et al.** [23] qui veulent standardiser les interfaces pour les bases de données NoSQL. Pour réaliser cette standardisation, ils introduisent un langage se voulant commun pour tous les systèmes NoSQL.

L'interface SOS (pour *Save Our System*) est un environnement de programmation où les bases de données non-relationnelles peuvent être uniformément définies par un logiciel.

L'interface SOS supporte déjà plusieurs systèmes NoSQL. L'homogénéité engendrée par une interface commune fait qu'il y a des compromis sur les performances actuelles. C'est un point à améliorer.

Cette interface se veut être une interface commune à chaque interface NoSQL, c'est un ORM pour le NoSQL tels que Hibernate [6] ou encore Doctrine [4].

D'ailleurs, **Doctrine** [4] et **Hibernate** [6] qui sont à la base des ORMs pour les bases de données relationnelles supportent également une base de données NoSQL tels que MongoDB [13]. Dans les cas où, comme ici, nous avons des ORMs pour du NoSQL, ils sont nommés ODMs (pour *Object Document Mapper*).

Si les ORMs sont une grande aide pour faciliter l'évolution des bases de données relationnelles, les ODMs ont la même vocation. Docrtine, Hibernate ou encore SOS sont tous les trois des interfaces utilisables pour les bases de données NoSQL. Ils facilitent le mapping entre le code et la base de données exactement comme le fait les ORMs.

Il est cependant utile de souligner que si le projet SOS a rencontré des problèmes de performances avec son interface [23], il en va sûrement de même pour les ODMs qui ont la même interface qu'un ORM.

Nous pouvons diminuer le temps d'arrêt de la base de données et du logiciel pour qu'il devienne insignifiant. **Blue-Green Deployment** est une solution proposée par **Fowler** [1]. C'est une solution pour déployer une nouvelle version avec possibilité de revenir rapidement en arrière en cas de problème. Le principe garantit une disponibilité maximale et est donc très pratique pour les sites webs par exemple.

Le principe tel que nous pouvons le voir sur la Figure 4.3 est très simple. Nous avons un environnement "prod" et une version "test". Sur l'environnement "test" nous mettons la nouvelle version. Une fois celle-ci déployée et validée, il ne reste plus qu'à changer le routeur pour qu'il pointe vers notre environnement

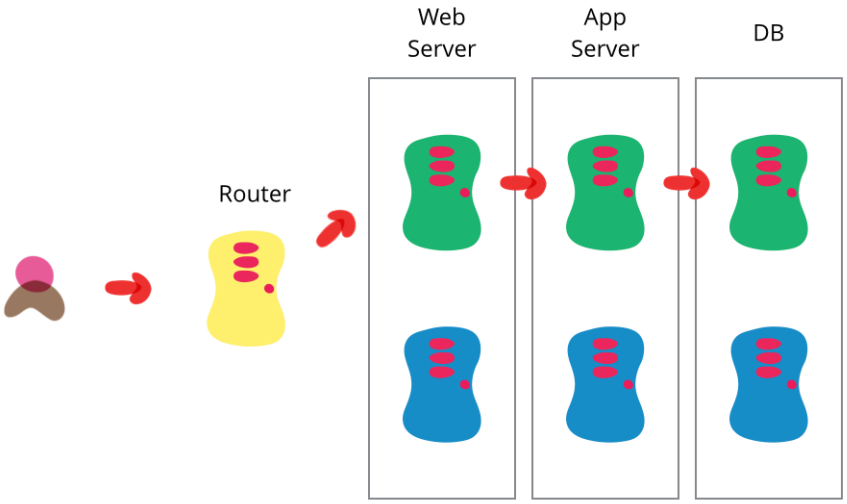


FIGURE 4.3 – Blue-Green Deployment [1]

de "test". Une fois l'opération réussie, l'environnement "test" devient celui de "prod" et vice-versa. Nous sommes prêts pour faire un nouveau déploiement.

Il est évident que nous pouvons combiner cette solution avec d'autres pour faciliter la mise en production. En combinant à d'autres techniques telles que **Liquibase** ou **FlywayDB** vues dans le chapitre précédent, il est possible de faciliter le retour en arrière. Il est également possible en utilisant les vues [22] de déployer de manière distincte le code de la base de données [45].
D'abord, nous ajoutons au schéma les nouveaux éléments. Dans ce processus, il faut veiller à ce que les nouveaux et anciens éléments (tables, colonnes, vues) soient synchronisés. Nous adaptons le programme pour qu'il prenne en compte le nouveau schéma. Enfin nous retirons les anciens éléments au schéma.

Remarquons que la technique proposée par **M. Fowler** (Blue-Green Deployment) [1] reste d'application pour tout type de bases de données.
La grosse différence entre les bases de données NoSQL et relationnelles étant l'absence de schéma, cela rend le déploiement séparé de la base de données et du code moins pertinent. En effet, en retirant la contrainte du schéma, nous pouvons simplement déployer notre nouveau code. Les données étant ce qui représente notre "schéma" dans le cas du NoSQL.

## 4.3 Plus rapide, plus découplé

Aujourd'hui, tout va de plus en plus vite. Nous avons aussi de plus en plus de problèmes de type big data. Avant, n'avoir qu'une application monolithique et sa base de données était la norme. Ce n'est plus le cas.

C'est pour cela que les sociétés se dirigent vers d'autres solutions pour les bases de données (sous-section 4.3.1) et qu'elles partent vers l'utilisation des micro-services (sous-section 4.3.2) au lieu d'une seule application monolithique.

### 4.3.1 Les bases de données

Le NoSQL n'est pas la solution ultime. Certaines société se sont vu contraintes de devoir migrer l'ensemble ou une partie de leur système du NoSQL vers du relationnel car ce type de base de données n'était pas adapté [32].

De plus, il existe énormément de systèmes NoSQL différents. Ces différences, comme nous avons déjà pu le voir dans la sous-section 3.2.2, ont donné naissance à des langages tel que SOS [23].

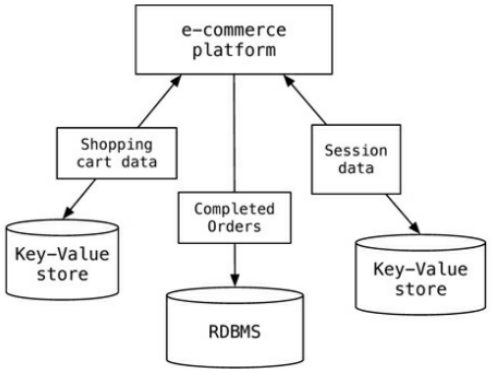


FIGURE 4.4 – Persistance polyglotte [60]

Les bases de données NoSQL ne sont pas la réponse ultime au stockage des données. **Sadalage et Fowler** [60] parlent de "*Polyglot Persistence*". La persistance polyglotte est l'utilisation de bases de données répondant au besoin spécifique à une fonctionnalité. Cela veut dire qu'une même application a potentiellement plusieurs espaces de stockage répondant à des problématiques différentes.

Comme dans l'exemple à la figure Figure 4.4. La plateforme e-commerce utilise des types de bases de données spécifiques à chaque fonctionnalité permettant une meilleure réponse aux utilisateurs.

Cette nouvelle problématique a vu l'émergence de "*polystore system*" [37, 44].

A cause de la grande différence de philosophie entre les types de bases de données, il reste beaucoup de chemin avant de voir un système permettant de stocker nos données dans la base de données la plus adaptée à la situation.

### 4.3.2 Les micro-services

Depuis plusieurs années, des solutions pour découper les applications monolithiques sont apparues. Dans un premier temps, il y a eu les architectures orien-

tées service (SOA) et désormais, nous voyons une grande tendance au micro-service. [11, 58]

Cette engouement s'explique par la difficulté de gérer une application monolithique. Lors de la modification d'une telle application, il est nécessaire de s'assurer des impacts potentiels à travers tout le système. Une fois le changement apporté, il faut tout "*builder*" et tout tester. [11] De plus, la tentation de faire un lien direct est forte quand la découpe n'est pas toujours claire. Les équipes doivent avoir une grande discipline pour garder les modules indépendants. L'utilisation de services rend plus explicites les interfaces et facilite l'encapsulation. [11]

Il existe des recherches pour identifier des fonctionnalités d'application orientées bases de données pour en faire d'éventuels services. **Del Grosso et al.** [41] proposent une telle approche.

L'identification est réalisée en observant les interactions entre l'application et la base de données via les requêtes extraites dynamiquement. D'autres approches utilisent l'interaction avec les utilisateurs pour identifier les services.

Une fois l'identification du ou des services réalisée, il est possible, avec d'autres approches, de migrer les fonctionnalités vers une application de type SOA. Il existe deux types d'approche pour migrer une application orientée base de données vers une application SOA. Un approche consiste en la migration d'une fonctionnalité à la fois. L'autre approche va, quand a elle, migrer toute l'application sous la forme d'un service.

# Conclusion

A travers cette recherche, nous avons procédé à un état de l'art afin de répondre à la question suivante : Comment faire évoluer une application ?

Pour y parvenir, nous avons parcouru les ouvrages ainsi que les outils traitant du sujet. Nous les avons recensés, analysés et comparés.

Afin de répondre à la question, nous nous sommes posés quatre sous-questions.

— Comment fonctionne le logiciel ?
— Si je modifie cette partie, qu'est-ce qui change ?
— Comment changer la base de données ?
— Comment adapter le programme ?

Chacune de ces questions correspond à un domaine particulier de l'informatique qui sont respectivement repris dans cette recherche comme étant : la rétro-ingénierie, l'analyse d'impacts, le changement de la base de données, l'adaptation du programme.

Dans notre recherche, nous avons tout d'abord vu comment récupérer la connaissance de l'application. Cette connaissance est dispersée à travers le code, la base de données, la documentation et toute personne travaillant avec le logiciel (développeurs ou utilisateurs).

Il existe de nombreuses techniques pour aider à comprendre l'application sur base de ces sources. La plupart de ces sources utilisent soit le code soit la base de données. La manière de visualiser les résultats de ces techniques sont soit un diagramme sous forme UML ou entité-relation, selon que nous traitions la base de données ou le code, soit une représentation graphique en utilisant, par exemple, DAHLIA, soit encore des statistiques ou des graphiques.

Peu de techniques ne se basent pas sur le code ou sur la base de données pour ressortir une partie de la connaissance de l'application. Parmi ces techniques, nous en avons qui simulent le comportement d'utilisateur afin d'identifier l'impact de changement de valeur à travers l'application. D'autres techniques se basent sur l'expérience des développeurs travaillant sur le logiciel pour en extraire l'architecture.

Nous avons pu constater que la recherche a beaucoup évolué depuis les débuts de la rétro-ingénierie. La recherche en matière de rétro-ingénierie s'est concentrée soit sur le code soit sur la base de données. La recherche sur la combinaison des deux reste, quant à elle, un champ peu exploré.

Deux autres aspects peu explorés qui méritent notre attention sont la documentation et l'expérience des personnes travaillant avec l'application tels que les développeurs et les utilisateurs.

Une fois la connaissance de l'application récupérée et avant d'apporter une modification sur celle-ci, nous nous sommes focalisés sur l'impact qu'une modification peut apporter au système.
Là où la rétro-ingénierie permet de récupérer la connaissance de l'application, l'analyse d'impacts est, quant à elle, là pour aider à garder la consistance de celle-ci après une modification. L'analyse d'impacts est là pour identifier les effets de bord éventuels sur le code ou la base de données.

Réaliser cette analyse manuellement est une tâche coûteuse et risquée. C'est la raison pour laquelle il existe des stratégies pour faciliter l'identification des effets de bord ainsi que des stratégies pour diminuer les impacts.
Afin de faciliter l'identification, les tests semblent être la meilleure solution, que ceux-ci soient au niveau du code ou de la base de données. Avoir des tests pertinents couvrant une bonne partie de l'application permet d'être averti rapidement de notre impact sur une autre partie de l'application. Si un test est en erreur après notre modification, nous savons que nous avons modifié le comportement de cette partie. Nous pouvons d'ailleurs lire de **Martin** que "*un code sans tests ne peut pas être propre*" [3].

Ces précautions peuvent ne pas suffire dans la mesure où des modifications peuvent passer au travers des mailles du filet de nos tests. Nous pouvons trouver dans la littérature plusieurs techniques pour aider l'identification des impacts. Certaines vont même jusqu'à proposer des modifications pertinentes des parties impactées. Parmi celles-ci, nous retiendrons la méthode "*what-if analyse*" qui est la plus représentée. Avec celle-ci, il est tout a fait envisageable de générer des tests automatiquement. Nous n'en avons pas parlé dans cette recherche, mais nous avons pu voir qu'il existait des travaux sur le sujet.

Maintenant que nous avons pris connaissance des impacts possibles lors d'une modification, nous nous sommes concentrés sur la manière de modifier la base de données. Nous nous sommes principalement focalisés sur les deux types de bases de données les plus utilisés aujourd'hui qui sont les bases de données relationnelles et NoSQLs.
Une grande différence entre les deux types de bases de données est la maturité des technologies. D'un côté, nous avons des bases de données très similaires d'un outil à l'autre. La recherche sur les bases de données relationnelles est très avancée. Il existe beaucoup d'ouvrages et d'outils la concernant.
De l'autre côté, nous avons une technologie plus récente. Si pas mal de recherches peuvent s'appliquer aux deux types, comme nous sommes dans deux paradigmes différents, tout n'est toutefois pas directement transposable. Une autre grosse différence est le fait que le NoSQL est "*schema-less*".

3. R.C. Martin. *Coder proprement.* Référence : programmation. Pearson, 2013, page 10.

Afin de pouvoir tout de même avoir une représentation de la structure des données dans les bases de données NoSQL, nous utilisons des schémas. Des règles ont été décrites pour passer d'un schéma conceptuel vers un schéma d'implémentation.

Comme le NoSQL est récent, chaque base de données NoSQL possède sa propre interface rendant l'utilisation spécifique et la migration plus complexe. Afin de faciliter et favoriser l'utilisation de ces types de bases de données, des tentatives de création d'interface commune existent. Certains outils se basent sur ces interfaces pour pouvoir interagir avec différentes basses de données NoSQL.

Qui dit nouveau type de bases de données, dit désir de l'utiliser. A la naissance du relationnel, des techniques pour migrer les données ont vu le jour. Celles-ci avaient en entrée, par exemple, des fichiers plats et permettaient la migration des données dans une base de données de type relationnel. Avec l'arrivée du NoSQL, la même chose se passe. De nombreuses techniques pour migrer automatiquement les données voient le jour. Elles font face à pas mal de problèmes dus au changement de paradigme. Il reste probablement encore du travail dans ce domaine.

Un constat lors de l'écriture de cette recherche est que, dans l'histoire du stockage des données, les anciennes manières de les stocker ont toujours été remplacées par la nouvelle si celle-ci était pertinente. A la différence des autres types de stockage, le NoSQL est complémentaire au relationnel. Ces deux types de bases de données répondent à des problématiques différentes. Aujourd'hui, nous n'avons que des recherches sur la migration du relationnel vers le NoSQL. Des recherches dans l'autre sens pourraient être réalisées, l'évolution du business est tel qu'il est possible que la solution du relationnel devienne plus appropriée.

La dernière des étapes dont nous nous sommes occupés est l'adaptation du programme. Adapter le programme passe par la mise à jour du code. La qualité de celui-ci détermine grandement la productivité d'une équipe. Son importance dans le développement de l'application est de plus en plus grande alors que les efforts à fournir pour le développement de la base de données sont de moins en moins importants. Il est donc essentiel de maintenir le code au mieux.

Pour garder une bonne qualité du code, les tests jouent un rôle très crucial. Il existe de nombreux frameworks et outils pour faciliter leur écriture. Si toutefois, nous n'arrivons pas à maintenir correctement le code, il existe des techniques pour s'occuper du code *legacy*.

L'évolution du code passe par celle de son interfaçage avec la base de données. Il n'existe pas beaucoup de recherches parlant de la co-évolution de la base de données et de l'application. Nous pouvons toute de même trouver plusieurs outils et techniques tels que les ORMs qui facilitent le mapping entre la base de données et le code. D'autres techniques existent pour, par exemple, éviter les temps d'arrêt des services lors du déploiement de la nouvelle version.

De nos jours, les sociétés se dirigent de plus en plus vers des solutions de microservices pour le développement de leur application. De même, elles privilégient l'utilisation d'ORMs et d'autres techniques du même genre pour diminuer leur dépendance à la base de données et ne plus faire de celle-ci qu'un outil comme un autre. De là, nous commençons à voir l'utilisation de multiples types de bases de

données pour un même projet. Cette nouvelle problématique a donné naissance à la persistance polyglotte.

D'un autre côté, en parcourant blogs et sites parlant de l'avenir de l'informatique et des bases de données [4] [5], nous allons vers le développement de celles-ci avec des problématiques de plus en plus complexe. A l'heure des micro-services, le cloud prend de plus en plus d'ampleur et avec lui, la question des bases de données distribuées. Smartphones et autres petits objets possèdent souvent une base de données. La recherche sur L'IoT (internet des objets) dans le cadre des bases de données mais aussi du développement en général a encore de beaux défits à relever. N'oublions pas aussi la place de mésintelligence artificielle que se soit pour extraire des informations de bases de données ou pour stocker et accéder aux quantités astronomiques de données générées par de tels systèmes.

Dans notre recherche, nous avons parcouru ce qui pouvait exister pour faire évoluer une application. Pour ce faire, nous nous sommes tout d'abord basés sur notre expérience personnelle. Celle-ci nous a permis d'avoir l'orientation de cette présente recherche. De là, nous avons commencé à parcourir les outils existants ainsi que la littérature pour diversifier notre recherche et approfondir notre réflexion afin d'avoir des comparaisons pertinentes.

Après exploration de tout ce dont nous avons parlé dans cette recherche, nous nous rendons compte qu'il est possible d'aller plus loin en étant plus systématique dans la recherche d'outils ou d'ouvrages. Nous n'avons pas recensé toutes les manières de faire évoluer une application. Dans chaque domaine, nous avons essayé de reprendre les plus pertinentes et celles encore utilisées ou récentes. Une recherche approfondie de tous les types de techniques pourrait faire ressortir des idées oubliées pouvant être remises en avant avec les avancées actuelles.

Dans le courant de cette recherche, nous avons pu parler d'approches qui, encore aujourd'hui, sont manuelles. Ces approches, comme "*Telling the story of the system*" [36], pourraient faire l'objet de futures recherches pour les automatiser. Un autre exemple est l'utilisation de la documentation. Celle-ci est reconnue pour ne pas être à jour. Pourtant, elle est bien souvent utile quand nous voulons savoir comment fonctionne l'application. Il reste encore beaucoup à faire du côté de la recherche pour définir des techniques de mise à jour de la documentation.

Dans cette recherche, nous avons tout aussi bien parcouru la littérature que des outils. Il en existe bien plus que ce que nous avons pu référencer. Pour répondre à une même problématique, tous ne font pas les mêmes choses ou pas de la même manière. Nous pourrions envisager une comparaison plus approfondie de ceux-ci.

Enfin, comme souligné ci-avant, nous avons utilisé notre expérience personnelle pour enrichir cette recherche afin d'amener un point de vue de praticien. Si nous pensons que notre expérience nous a permis d'apporter un plus, celui-ci

4. https://internetofthingsagenda.techtarget.com/blog/IoT-Agenda/Challenges-of-data-management-in-the-internet-of-things
5. https://dawn.cs.stanford.edu/2018/04/11/db-community/

n'est pas suffisant. Il serait intéressant d'aller auprès des sociétés afin de constater leurs problématiques au quotidien dans la maintenance de leurs applications. Il est également envisageable d'aller auprès de ces sociétés avec des idées et outils venant de la recherche pour évaluer leur utilité.

Nous nous sommes rendus compte de la pertinence de la rencontre entre la recherche et la pratique.

De fait, la présente recherche nous a apporté beaucoup dans notre pratique. Elle nous a permis d'élargir nos connaissances au-delà des outils et des techniques couramment utilisés. Nous avons pu développer notre esprit critique, ce qui nous aide quand nous devons choisir de nouveaux outils, framewoks ou lors de toutes autres décisions. Nous avons également pris conscience de l'importance du code bien fait. Si avoir une application bien testée avec une bonne architecture et un code clair était déjà une évidence avant, c'est en lisant les livres de **Feathers** [36] et **Martin** [51] que nous avons pu mettre des mots sur la raison de cette évidence.

# Bibliographie


[1] Bliki : Bluegreendeployment. https://martinfowler.com/bliki/BlueGreenDeployment.html. Accédé le 26/04/2019.

[2] Convert-in. http://www.convert-in.com/. Accédé le 27/06/2019.

[3] Db main. https://www.rever.eu/fr/db-main/. Accédé le 05/05/2019.

[4] Doctrine. https://www.doctrine-project.org. Accédé le 20/06/2019.

[5] Flywaydb. https://flywaydb.org/. Accédé le 20/06/2019.

[6] Hibernate. https://hibernate.org/. Accédé le 20/06/2019.

[7] Ispirer. http://www.ispirer.fr/. Accédé le 27/06/2019.

[8] Jetbrains. https://www.jetbrains.com. Accédé le 04/05/2019.

[9] Liquibase. https://www.liquibase.org/. Accédé le 20/06/2019.

[10] Liquibase-hibernate. https://github.com/liquibase/liquibase-hibernate/. Accédé le 20/06/2019.

[11] Microservices. https://martinfowler.com/articles/microservices.html. Accédé le 26/04/2019.

[12] Mockito. https://site.mockito.org/. Accédé le 21/07/2019.

[13] Mongodb. https://www.mongodb.com/. Accédé le 19/05/2019.

[14] Smo. http://yellowstone.cs.ucla.edu/schema-evolution/index.php/SMO. Accédé le 26/04/2019.

[15] Software engineering mit easycode. http://www.easycode.de/. Accédé le 26/04/2019.

[16] Sonarsource. https://www.sonarsource.com. Accédé le 04/05/2019.

[17] Staruml 3. http://staruml.io/. Accédé le 26/04/2019.

[18] Visual paradigm. https://www.visual-paradigm.com/. Accédé le 04/05/2019.

[19] 9 best reverse engineering tools for 2019. https://www.apriorit.com/dev-blog/366-software-reverse-engineering-tools, Dec 2018. Accédé le 26/04/2019.

[20] Fatma Abdelhédi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Processus de transformation mda d'un schéma conceptuel de données en un schéma logique nosql. In *INFORSID*, 2016.

[21] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Wafa : Fine-grained dynamic analysis of web applications. *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 141–150, 2009.

[22] Scott W. Ambler and Pramodkumar J. Sadalage. Refactoring databases : Evolutionary database design. 2006.

[23] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems : The sos platform. In *CAiSE*, 2012.

[24] Andreas Behm, Andreas Geppert, and Klaus R. Dittrich. On the migration of relational schemas and data to object-orienteddatabase systems. 1997.

[25] Alessandro Bianchi, Danilo Caivano, and Giuseppe Visaggio. Method and process for iterative reengineering of data in a legacy system. In *WCRE*, 2000.

[26] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco : A model driven reverse engineering framework. *Information & Software Technology*, 56 :1012–1032, 2014.

[27] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. *Future of Software Engineering (FOSE '07)*, pages 326–341, 2007.

[28] Shi-Kuo Chang, Vincenzo Deufemia, Giuseppe Polese, and Mario Vacca. A logic framework to support database refactoring. In *International Conference on Database and Expert Systems Applications*, pages 509–518. Springer, 2007.

[29] Elliot Chikofsky. The necessity of data reverse engineering. *Preface to Data Reverse Engineering : Slaying the Legacy Dragon. McGraw-Hill, New York*, 1996.

[30] Anthony Cleve. *Program analysis and transformation for data-intensive system evolution.* PhD thesis, 2010.

[31] Anthony Cleve, Jean-Roch Meurisse, and Jean-Luc Hainaut. Database semantics recovery through analysis of dynamic sql statements. *J. Data Semantics*, 15 :130–157, 2011.

[32] E. F. Codd. A relational model for large shared data banks. 1970.

[33] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22 :73–98, 2012.

[34] Leonardo C. da Rocha, Fernando Vale, Elder Cirilo, Dárlinton Barbosa, and Fernando Mourão. A framework for migrating relational datasets to nosql1. In *ICCS*, 2015.

[35] Kathi Hogshead Davis and Peter H. Aiken. Data reverse engineering : A historical survey. In *WCRE*, 2000.

[36] Michael Feathers. *Working Effectively with Legacy Code.* Prentice Hall Professional, 2004.

[37] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron J. Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Timothy G. Mattson, and Michael Stonebraker. The bigdawg polystore system and architecture. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.

[38] Spyridon K. Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. 2009.

[39] Sunita Ghotiya, Juhi Mandal, and Saravanakumar Kandasamy. Migration from relational to nosql database. 2017.

[40] Katarina Grolinger and Miriam A. M. Capretz. A unit test approach for database schema evolution. *Information & Software Technology*, 53 :159–170, 2011.

[41] Concettina Del Grosso, Massimiliano Di Penta, and Ignacio García Rodríguez de Guzmán. An approach for mining services in database oriented applications. *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 287–296, 2007.

[42] Eduardo Guerra and Maurício Aniche. Chapter 9 - achieving quality on software design through test-driven development. In Ivan Mistrik, Richard Soley, Nour Ali, John Grundy, and Bedir Tekinerdogan, editors, *Software Quality Assurance*, pages 201 – 220. Morgan Kaufmann, Boston, 2016.

[43] Robin Hecht and Stefan Jablonski. Nosql evaluation : A use case oriented survey. *2011 International Conference on Cloud and Service Computing*, pages 336–341, 2011.

[44] Moditha Hewasinghage, Nacéra Bennacer Seghouani, and Francesca Bugiotti. Modeling strategies for storing data in distributed heterogeneous nosql databases. In *ER*, 2018.

[45] Thorben Janssen, Rafael Ponte, Thorben Janssen, Tracy Woo, and Thorben Janssen. Update your database schema without downtime. `https://thoughts-on-java.org/update-database-schema-without-downtime/`, Jul 2018. Accédé le 26/04/2019.

[46] Chao-Hsien Lee and Yu-Lin Zheng. Automatic sql-to-nosql schema transformation over the mysql and hbase databases. *2015 IEEE International Conference on Consumer Electronics - Taiwan*, pages 426–427, 2015.

[47] Dachuan Liang, Yunzhen Lin, and Guiguang Ding. Mid-model design used in model transition and data migration between relational databases and nosql databases. *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 866–869, 2015.

[48] Chang-Yang Lin. Migrating to relational systems : Problems, methods, and strategies. 2008.

[49] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Aiding maintenance of database applications through extracting attribute dependency graph. *J. Database Manag.*, 24 :20–35, 2013.

[50] Meurice Loup. *Analyzing, Understanding and Supporting the Evolution of Dynamic and Heterogeneous Data-Intensive Software Systems*. PhD thesis, UNamur. Faculté d'informatique, 2017.

[51] R.C. Martin. *Coder proprement*. Référence : programmation. Pearson, 2013.

[52] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 451–460, 2008.

[53] Loup Meurice, Csaba Levente Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 262–273, 2016.

[54] Noughi Nesrine. *Understanding Data-Intensive Systems Through The Analysis of SQL Execution Traces*. PhD thesis, UNamur. Faculté d'informatique, 2018.

[55] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. What-if analysis for data warehouse evolution. In *DaWaK*, 2007.

[56] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. Hecataeus : Regulating schema evolution. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 1181–1184, 2010.

[57] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *ESEC/SIGSOFT FSE*, 2013.

[58] F. Rademacher, S. Sachweh, and A. Zündorf. Differences between model-driven development of service-oriented and microservice architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 38–45, April 2017.

[59] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Record*, 35 :30–31, 2006.

[60] Pramod Sadalage and Martin Fowler. Nosql distilled : A brief guide to the emerging world of polyglot persistence. 2012.

[61] Stefanie Scherzinger, Thomas Cerqueus, and Eduardo Cunha de Almeida. Controvol : A framework for controlled schema evolution in nosql application development. *2015 IEEE 31st International Conference on Data Engineering*, pages 1464–1467, 2015.

[62] Liana Stanescu, Marius Brezovan, and Dumitru Dan Burdescu. Automatic mapping of mysql databases to nosql mongodb. *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 837–840, 2016.

[63] Bipin Upadhyaya, Ran Tang, and Ying Zou. An approach for mining service composition patterns from execution logs. *Journal of Software : Evolution and Process*, 25(8) :841–870, 2013.

[64] J Yoo, Ki-Hoon Lee, and Y.-H Jeon. Migration from rdbms to nosql using column-level denormalization and atomic aggregates. *Journal of Information Science and Engineering*, 34 :243–259, 01 2018.

[65] Gansen Zhao, Weichai Huang, Shunlin Liang, and Yanzhen Tang. Modeling mongodb with relational model. *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 115–121, 2013.

[66] Gansen Zhao, Qiaoying Lin, Libo Li, and Zijing Li. Schema conversion model of sql database to nosql. *2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 355–362, 2014.